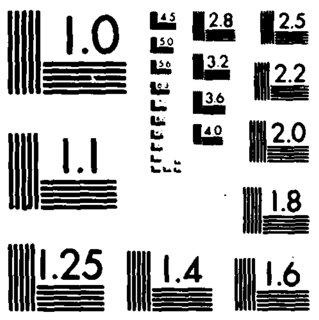
AD-A092 691 STANFORD UNIV CA DEPT OF OPERATIONS RESEARCH F/G 12/1
AN ADDITIVE ALGORITHM FOR THE MULTIPLE CHOICE INTEGER PROGRAM.(U)
OCT 80 J C BEAN N00014-76-C-0418
UNCLASSIFIED TR-96 NL

1 of 2
AD A
092691


1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

AN ADDITIVE ALGORITHM FOR THE MULTIPLE CHOICE INTEGER PROGRAM

by

JAMES C. BEAN

TECHNICAL REPORT NO. 96
OCTOBER 1980

PREPARED UNDER CONTRACT
N00014-76-C-0418 (NR-047-061)
FOR THE OFFICE OF NAVAL RESEARCH

Frederick S. Hillier, Project Director

This research was supported in part by National Science Foundation Grant MCS76-81259 - Department of Operations Research, Stanford University and issued as Technical Report No. 80-26.

DEPARTMENT OF OPERATIONS RESEARCH
STANFORD UNIVERSITY
STANFORD, CALIFORNIA

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Most techniques in the field of mathematical programming deal with the problem of making choices in a world where scarcity limits the set of options that are available. Mathematical programming attempts, quantitatively, to determine the best of these available options. The greatest emphasis and success in this field has come in problems where the values from which to choose form a continuum. The outstanding example is when the set of feasible choices can be described by a set of linear *inequalities* and the *quality* of any choice can be determined by evaluating a linear function. For this linear programming problem Dantzig's simplex method allows the solution of large and complex problems.

Many problems of interest require choice from a countable set, most commonly a subset of the non–negative integers. Though on the surface it appears this type of problem should be much simpler due to the more limited set of choices, this is not at all the case, even in this linear example. With continuous variables, the geometry of the linear problem allows all but a finite number of the uncountably many possible solutions to be eliminated from contention without analysis. (These remaining possible solutions are extreme points of the feasible region from which the set of all feasible solutions can be generated by convex combinations.) Furthermore, there is a graceful way of quickly analyzing these remaining points when they are not too numerous. When the choice variables must take integer values, in all but the binary case, the best solution need not be an extreme point of the feasible set. In the binary case all feasible solutions are at extreme points so limiting investigation to extreme points gains nothing except if they are few in number. As a result, relatively small linear problems with integral choices can take a long time to solve.

Attacks on this class of problems have come from many directions, most notably cutting plane methods and branch-and-bound methods. Geoffrion and Marsten [Geoffrion & Marsten, 1972-1] have written an excellent survey of these and other techniques. An important sub-class of branch-and-bound methods, binary additive type techniques, will be the framework of the algorithm presented in this dissertation.

The problem dealt with in this dissertation will be a special case of the integer programming problem with binary variables. The specific problem is one called the Multiple Choice Integer Program (MCIP) in which the variables can be partitioned (using the classical definition of "partition") such that exactly one variable in each partitioning set has the value one in any feasible solution and all other variables are zero. A mathematical formulation of the problem is

$$\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{i=1}^{m}\sum_{j=1}^{n_i} c_{ij}x_{ij} & \\
\text{subject to:} & b + Ax \geq 0 & \qquad\qquad (MCIP) \\
 & \displaystyle\sum_{j=1}^{n_i} x_{ij} = 1 \qquad , i = 1, 2, \ldots, m & \\
 & x_{ij} \in \{0, 1\}. &
\end{array}$$

The details of this formulation will be presented in Chapter 2.

Chapter 2 will discuss the history of this problem and solution techniques developed for it and similar problems. Chapter 3 will present discussion of additive type algorithms and the algorithm proposed here. It will also include definitions and notation that will be used thereafter. Applications of this algorithm follow in Chapter 4 with worked examples for three problems. In Chapter 5 it will be proven that the algorithm will correctly solve any problem of this type in finite time. In Chapter 6 Geoffrion's surrogate constraints are revised to take advantage of the additional structure in this problem, relative to general binary programming. An analysis of worst case performance of the algorithm is included in Chapter 7 along with some techniques to speed up the algorithm, a heuristic version of the algorithm and results of experimental runs on two classes of problems. Finally, Chapter 8 will state conclusions and areas for future research.

CHAPTER 2

HISTORY OF THE MULTIPLE CHOICE INTEGER PROGRAM

To grasp the value of the MCIP it is necessary to examine the history of problems of this type. Certainly the oldest related problem is the classical assignment problem.

- **The Classical Assignment Problem**

The assignment problem was treated during the earliest years of mathematical programming, even before the second world war (see [Dantzig, 1977-1]). The assignment problem is a special case of the transportation problem, perhaps the most explored specially structured problem in mathematical programming.

The assignment problem can be described as follows. Given m workers and m tasks it is necessary to assign each worker a task. Each worker/task pair is associated with a cost. One possible interpretation of the cost is the time required by that worker to complete that task. If the tasks are indexed by i, for $i \in \{1, 2, \ldots, m\}$, and if the workers are indexed by j, for $j \in \{1, 2, \ldots, m\}$, let the cost associated with the task/worker pair (i, j) be c_{ij}. The problem will be formulated with m^2 variables

$$\{x_{ij} : i = 1, 2, \ldots, m; j = 1, 2, \ldots, m\}$$

such that x_{ij} is one if task i is assigned to worker j and zero if it is not. There must be exactly one job assigned each worker and each job must be assigned to exactly one worker. The mathematical formulation of the assignment problem is then

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{m}\sum_{j=1}^{m} c_{ij}x_{ij} & \\ \text{subject to:} & \sum_{i=1}^{m} x_{ij} = 1 & , j = 1, 2, \ldots, m \\ & \sum_{j=1}^{m} x_{ij} = 1 & , i = 1, 2, \ldots, m \\ & x_{ij} \in \{0, 1\} & , \forall i; \forall j. \end{array} \qquad (ASN)$$

There are two classical techniques for solving this problem: the transportation variant of the simplex algorithm and Kuhn's Hungarian Method. The former is based on the fact that the assignment problem is a special case of transportation problem. It uses the fact that, when solved by the simplex method, all bases for the transportation problem are triangular. This saves a great deal of the effort necessary in the general simplex algorithm. Details on this method can be found in [Dantzig, 1963–1].

The Hungarian method is based on a combinatorial argument. It begins by recognizing that if a constant is added to all the costs corresponding to one worker or all the costs corresponding to one task, the set of optimal assignments does not change. (A similar though more limited observation about MCIP will become important in the next chapter.) Details on the Hungarian algorithm can be found in [Kuhn, 1955–1].

- **The Generalized Assignment Problem**

The classical assignment problem requires that each worker be assigned only one task. In the generalized assignment problem this requirement is relaxed. Here a worker can be assigned multiple tasks so long as they do not use more of some scarce resource than is available. For example, the scarce resource may be time. Assume a worker works eight hours a day. Then in one day he or she could be assigned two jobs taking three hours each and one taking two hours, but could not be assigned three jobs taking three hours each. Formally this problem can be stated as

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} \\ \text{subject to:} \quad & \sum_{i=1}^{m} r_{ij}x_{ij} \leq b_j \quad , j = 1,2,\ldots,n \\ & \sum_{j=1}^{n} x_{ij} = 1 \quad , i = 1,2,\ldots,m \\ & x_{ij} \in \{0,1\} \quad \forall i; \forall j, \end{aligned} \qquad (GASN)$$

where r_{ij} could be interpreted as the number of hours required by worker j to accomplish task i, c_{ij} the expenses per hour if worker j is assigned to task i and b_j as the total number of hours worker j works in the planning period.

Ross and Soland have developed a method for solving this problem that is presented in [Ross & Soland, 1973–1]. This is a conventional branch-and-bound procedure except that lower bounds are partially obtained by solving binary knapsack problems rather than linear programs. These knapsack problems can be solved very quickly leading to the efficiency cited in [Ross & Soland, 1973–1].

This technique looks at GASN as m problems, each having as its variables those contained in a constraint of the type

$$\sum_{j=1}^{n} x_{ij} = 1 \quad , i = 1, 2, \ldots, m. \qquad (GUB)$$

Each constraint of this type is referred to as a generalized upper bound. Recall that a standard upper bound for some variable y is stated $y \leq u$, where u is a constant. Dividing by u (assuming it is greater than zero) and scaling the variable y, this becomes $y \leq 1$. An upper bound becomes generalized when a sum of variables are limited by the bound, for example,

$$y_1 + y_2 + \cdots + y_k \leq 1.$$

When a slack variable is added to this constraint it takes the form of GUB. The equality form of this constraint is what shall be meant by "generalized upper bound" for the remainder of this dissertation.

A problem may have many such constraints. In GASN there are m GUB–constraints. The set of variables in one GUB–constraint are referred to as a GUB–set. In the case of GASN these sets partition the variables, that is, each variable in the problem is in exactly one GUB–set.

In GASN each variable is binary, that is, it can take on only the values zero or one. Combined with the GUB–structure, this implies that exactly one variable in each GUB–set must have the value one in any feasible solution. All other variables must have the value zero. This characteristic is called the multiple choice property.

The procedure of Ross and Soland begins by ordering the variables in each GUB–set according to their costs, c_{ij}, lowest first. The variable labelled x_{i1} has the lowest cost of any variable in GUB–set i. The variable x_{in} has the highest cost in GUB–set i. By the multiple choice characteristic of this problem, finding a feasible solution involves choosing one variable from each GUB–set to be set to one. Ross and Soland begin their procedure by choosing the variables $\{ x_{i1} \text{ for } i = 1, 2, \ldots, m \}$. These are the variables having lowest cost in each GUB–set. If this choice is feasible it is optimal since it cannot have greater cost than any other set of choices. Except for trivial problems, this choice of variables will not be feasible in the resource constraints. The procedure then uses the branch–and–bound attack mentioned to find the lowest cost choice that is feasible. Additional details on this procedure can be found in [Ross & Soland, 1973–1].

The concepts of generalized upper bounds, multiple choice constraints and cost ordering are presented above in some detail because they also play an important part in the algorithm presented in this dissertation. Though this algorithm is not based on the algorithm of Ross and Soland, it is of interest that similar concepts have been used to attack these related problems.

- **Multiple Choice Program**

The analysis of multiple choice type constraints began with W. C. Healy, Jr. in a 1964 paper [Healy, 1964–1]. He formulated the problem

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{m}\sum_{j=1}^{n_i} c_{ij}x_{ij} \\ \text{subject to:} & b + Ax \geq 0 \\ & \sum_{j=1}^{n_i} x_{ij} = 1 \quad ,i = 2,3,\ldots,m \\ & x_{ij} \in \{0,1\} \quad ,i = 2,3,\ldots,m, \forall j \\ & x_{1j} \geq 0 \quad ,j = 1,2,\ldots,n_1 \end{array} \qquad (MCP)$$

where n_i is the number of variables in the i^{th} GUB–set, for $i = 1,2,\ldots,m$. Note that in this problem there is a set of variables, indexed 1, that are not limited by a GUB–constraint and are continuous and non–negative rather than binary. This is a generalization of the MCIP.

The solution method presented by Healy is as follows. The problem is solved as a linear program with the integrality constraints relaxed. In all but trivial problems, the resulting solution will violate these integrality requirements. Then in each GUB–set a combination of zeros and ones is found for the non-integral variables that causes the least increase in the objective value from the optimal value of the relaxed LP, given the variables that were integral in the LP solution. This combination most probably will not be feasible in the general constraints. If this is the case, a constraint is added to the problem restricting the objective value to be not less than the optimal value of the last LP plus the minimal increase found above. The linear program is then resolved. The new LP solution may have some non-integral values, though they may be for a different set of variables than in the first run. The process then continues as above. This process is repeated until a feasible integral solution is found.

Unfortunately, this procedure is not guaranteed to find an optimal, or even feasible, solution. When there exists a non-integral solution with the same objective value as the optimal integral solution the algorithm can be shown to fail to locate the correct solution. It does not necessarily find the optimal solution in other cases either.

- **Generalized Upper Bounding Problem**

At about the same time Healy developed the MCP, George B. Dantzig and Richard Van Slyke developed a technique for solving a similar continuous variable problem, the generalized upper bounding problem ([Dantzig & Van Slyke, 1964-1]). This problem is precisely the one solved by Healy when integrality restrictions are relaxed. This problem can be stated as

$$\begin{array}{lll} \text{minimize} & \displaystyle\sum_{i=1}^{m}\sum_{j=1}^{n_i} c_{ij}x_{ij} & \\ \text{subject to:} & b + Ax \geq 0 & \\ & \displaystyle\sum_{j=1}^{n_i} x_{ij} = 1 \quad , i = 2, 3, \ldots, m & (GUBP) \\ & x_{ij} \geq 0 \quad , \forall i; \forall j. & \end{array}$$

It was here that the term generalized upper bounding was first used.

This technique sets up the problem as a linear program, then reduces the usual $\kappa + m$ order basis to a working basis of order m by a clever transformation, where κ is the number of constraints in $b + Ax \geq 0$. If m is large compared to κ, this can save a great deal of work when inverting the basis.

- **The Multiple Choice Knapsack Problem**

The multiple choice knapsack problem can be stated as

$$\begin{array}{lll} \text{minimize} & \displaystyle\sum_{i=1}^{m}\sum_{j=1}^{n_i} c_{ij}x_{ij} & \\ \text{subject to:} & \displaystyle\sum_{i=1}^{m}\sum_{j=1}^{n_i} a_{ij}x_{ij} \leq b & (MCKP) \\ & \displaystyle\sum_{j=1}^{n_i} x_{ij} = 1 \quad , i = 1, 2, \ldots, m & \\ & x_{ij} \in \{0, 1\} \quad \forall i; \forall j. & \end{array}$$

This problem is similar to the MCP, differing in that the multiple choice constraints are exhaustive and that there is only one general constraint.

The algorithm presented in [Sinha & Zoltners, 1979–2] appears to solve problems of this type very efficiently. It begins by analyzing relationships between the coefficients $\{c_{ij}\}$ and $\{a_{ij}\}$. Under certain conditions, some variables may be eliminated from consideration. Next a branch–and–bound procedure is employed that uses the Beale–Tomlin multiple choice dichotomy [Beale & Tomlin, 1969–1]. Let G be a GUB–set of variables. Then exactly one element of the set G has the value one in any feasible solution. Partition G into two subsets so that $G = H \bigcup H^c$. Then the multiple choice dichotomy states that one element of H has the value one, or one element of H^c has the value one, where the "or" is exclusive.

- **The Multiple Choice Integer Program**

The multiple choice integer program is the problem dealt with in this dissertation. It can be stated as

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{m}\sum_{j=1}^{n_i} c_{ij}x_{ij} \\ \text{subject to:} & b + Ax \geq 0 \\ & \sum_{j=1}^{n_i} x_{ij} = 1 \qquad ,i = 1,2,\ldots,m \\ & x_{ij} \in \{0,1\} \quad ,\forall i; \forall j. \end{array} \qquad (MCIP)$$

The non–GUB constraints are more general in MCIP than GASN. The resource constraints in GASN each contain only variables associated with one particular worker. The general constraints in MCIP can contain any linear inequality. Consequently, GASN is a special case of MCIP.

The multiple choice program is more general than MCIP. It allows continuous variables that are not GUB constrained.

The multiple choice knapsack problem is a special case of the MCIP. The only difference here is that MCKP allows only one general constraint whereas MCIP allows any number.

Peter Mevert and Uwe Suhl in [Mevert & Suhl, 1977–1] present an algorithm to solve this problem. They use a branch-and-bound procedure, as does the algorithm presented in this dissertation.

Another algorithm for solving MCIP was developed by P. Sinha and Andris Zoltners,[Sinha & Zoltners, 1979–1]. They also use a branch-and-bound procedure, but one that is less similar to the algorithm in this dissertation than the work of Mevert and Suhl. The Beale–Tomlin multiple choice dichotomy is employed rather than a branching scheme reminiscient of Balas' additive algorithm [Balas, 1965–1]. The general constraints are reduced to a single surrogate constraint and the algorithm in [Sinha & Zoltners, 1979–2] applied. The method they use to construct the single constraint is similar to the method presented in the surrogate constraint section of Chapter 6.

Further discussion of these algorithms will appear in Chapter 7.

CHAPTER 3

FORMAL STATEMENT OF THE ALGORITHM

A formal statement of the problem being considered is:

$$\begin{aligned}
&\text{minimize} && \sum_{i=1}^{m}\sum_{j=1}^{n_i} c_{ij}x_{ij} \\
&\text{subject to:} && b + Ax \geq 0 \\
& && \sum_{j=1}^{n_i} x_{ij} = 1 \quad ,i = 1,2,\ldots,m \\
& && x_{ij} \in \{0,1\}.
\end{aligned} \qquad (MCIP)$$

MCIP is a general 0—1 integer program with exhaustive multiple choice constraints. That is, each binary variable must belong to exactly one subset of the variables such that for any feasible solution exactly one variable in each subset has the value 1 and the rest have the value 0.

To understand the concept behind this algorithm, consider the tree for a six variable binary problem in figure 3-1 (at the end of the chapter). At each node, choosing the upper branch when leaving that node indicates that the corresponding variable is being set to 1 and taking the lower branch indicates it is 0. This tree shows the $2^6 = 64$ branches that must be enumerated (perhaps implicitly) to solve a problem of this size. Now assume the constraints on the problem contain the following:

$$x_1 + x_2 + x_3 = 1; \qquad (GUB1)$$

$$x_4 + x_5 + x_6 = 1. \qquad (GUB2)$$

There are only three sets of values for the variables x_1, x_2 and x_3 that satisfy (GUB1). They are

$$x_1 = 1, \quad x_2 = 0, \quad x_3 = 0;$$

$$x_1 = 0, \quad x_2 = 1, \quad x_3 = 0;$$

$$x_1 = 0, \quad x_2 = 0, \quad x_3 = 1.$$

Further, there are only three sets of values for x_4, x_5 and x_6 that satisfy (GUB2), namely,

$$x_4 = 1, \quad x_5 = 0 \quad x_6 = 0;$$

$$x_4 = 0, \quad x_5 = 1, \quad x_6 = 0;$$

$$x_4 = 0, \quad x_5 = 0, \quad x_6 = 1.$$

These GUB–feasible combinations are indicated in figure 3–1 by the broken line tree. Note that of the 64 branches for the general six variable problem, only nine satisfy the GUB–constraints.

This algorithm enumerates the smaller tree rather than the larger leading to savings from two sources. The primary savings results from the fact that there are many fewer branches to enumerate. Beyond this, there are now fewer constraints to consider since the GUB constraints are now handled implicitly. No potential solution will have to be tested for GUB–feasibility since only GUB–feasible solutions are considered.

• Branch-and-Bound Algorithms

The structure of Balas' additive algorithm for binary programs, a special type of branch-and-bound procedure, will serve as a framework for this algorithm. Figure 3-2 shows the flowchart of a general branch-and-bound algorithm. The concept is as follows. If the problem can be solved by inspection, do so. If not, separate the set of possible solutions into two or more groups. For example, one common method is to choose some variable, say x_i, and consider all solutions that have $x_i = 0$, and all solutions that have $x_i = 1$ as the two groups. The result is a multiplicity of smaller problems called "descendants." When all of these smaller problems are solved, the best of the candidate optimal solutions is optimal in the original problem, where a candidate optimal solution is constructed by combining the variables fixed by S and its optimal completion to form a solution to the original problem. This attack is often referred to as "divide and conquer."

What characterizes the additive algorithm is: 1) how the separation is accomplished and 2) how descendants are solved. The separation is done by choosing some subset of the variables and constructing descendants by fixing these variables at all possible sets of values they can take on. When specific values are assigned to a set of variables it is called a partial solution, usually denoted S. The number of descendants is the number of different sets of values these variables can have. For example, consider the problem

$$\begin{array}{ll} \text{minimize} & cx \\ \text{subject to:} & b + Ax \geq 0 \\ & x_j \in \{0,1\} \quad , j = 1, 2, \ldots, n. \end{array}$$

One possible separation of this problem would be into four descendants,

1) $\min\{cx{:}b + Ax \geq 0, x_j \in \{0,1\}, x_1 = 0, x_2 = 0\}$

2) $\min\{cx{:}b + Ax \geq 0, x_j \in \{0,1\}, x_1 = 0, x_2 = 1\}$

3) $\min\{cx{:}b + Ax \geq 0, x_j \in \{0,1\}, x_1 = 1, x_2 = 0\}$

4) $\min\{cx{:}b + Ax \geq 0, x_j \in \{0,1\}, x_1 = 1, x_2 = 1\}$.

The partial solutions for these four descendants are, respectively,

$$S_1 = \{x_1 = 0, x_2 = 0\},$$

$$S_2 = \{x_1 = 0, x_2 = 1\},$$

$$S_3 = \{x_1 = 1, x_2 = 0\},$$

$$S_4 = \{x_1 = 1, x_2 = 1\}.$$

Hopefully, the added information of the values for x_1 and x_2 will make each of these descendants much easier to solve than the original problem. The variables that are not assigned values in a descendant are called "free" variables, in this example $x_3, x_4, \ldots, x_n$.

The additive algorithm employs a recursive structure to solve descendants. Each descendant is viewed as an original problem and, if it cannot be solved by inspection, the set of possible solutions is separated again. An attempt is made to solve all second generation descendants. If any are difficult, that problem will be separated again. This process must be finite so long as the separations are non-empty and disjoint. Since there are only a finite number of solutions to be considered, eventually separations can be made so fine that each descendant has only one possible solution. At this point the optimal value for each descendant is obvious. This worst case is called exhaustive enumeration since every solution is examined explicitly.

It is not desirable to terminate by considering all possible solutions explicitly. If n is the number of variables, the number of possible solutions is 2^n. If n is 30 there are over a billion possible solutions. The efficiency of this type of algorithm comes from looking at only a few solutions explicitly while examining the rest implicitly, that is, to infer that they need not be considered from information gained from those few solutions explicitly examined. This inference takes place when descendant problems are analyzed to determine if they can be solved by inspection. The more clever the algorithm is at solving problems by inspection, the more efficient the algorithm. Chapter 6 is devoted to this question.

At any point, the best solution that has been discovered to that time is called the "incumbent." When the inspection finds a solution better than the incumbent, it takes the place of the incumbent.

- **Augmentation**

Augmenting rules determine how a problem should be separated into descendants and the order in which the descendants should be examined. For the algorithm presented here, the separation used will be that suggested by the broken–line tree in figure 3–1. When a partial solution is augmented, a free GUB–set will be chosen and the remaining problem separated into descendants each corresponding to a different element of this GUB–set being set to 1. The number of descendants is equal to the number of elements in the GUB–set chosen,n_i.

In order to determine which descendant to analyze next, a cost ordering procedure will be implemented. When setting up the algorithm, the variables in each GUB–set will be ordered by cost coefficient, lowest first. After this cost ordering the lowest cost solution that satisfies the GUB–constraints is found by choosing the first element in each GUB–set. If this solution is feasible for the general constraints, it is optimal and the problem is solved. If not, then the feasible region will be separated and one of the descendants attacked. The separation will be done as described above, choosing the GUB–set with the lowest cost element in the problem. Of these descendants, the one attacked first will be that corresponding to a 1 at this element of lowest cost. A formal statement of general augmentation rules will follow shortly.

- **Backtracking**

When a descendant problem is solved it is necessary to know what descendants remain and which should be attacked next. The procedure for accomplishing this is backtracking. When one descendant is solved, a logical course to follow is to attack another descendant of that same separation. It turns out, however, that for this algorithm and the MCIP this is not the most efficient rule. When solving any descendant, information is gained that may indicate that the next descendant that should be solved is a descendant of a different separation. This algorithm allows the attack of this alternative descendant. Allowing this flexibility, it becomes difficult to keep track of which descendants have been enumerated. In the tree of a binary additive algorithm (figure 3–1) each node has only two arcs leaving it. Hence, when the algorithm retreats to a node it only needs to know if the other arc has been investigated. The tree for the algorithm presented here can have any number of arcs leaving any node. Just recording which arcs have been investigated requires a great deal of storage. By using a slightly more complicated recording procedure, the algorithm gains the ability to choose descendants from other separations with little storage problem. This flexibility allows quicker convergence on the optimal solution.

- **Definitions**

Define a **position** as a vector $u \in Z^m$ indicating which element in each GUB–set has value 1. For example, if and $u(1) = 2$, then $x_{12} = 1$ and all other elements of GUB–set 1 are 0.

As the algorithm progresses, at certain times it is deemed necessary to store the current position. At some future point in the progress of the algorithm, it will return to this position so that it can strike out in another direction.

Consider climbers ascending a mountain never before climbed. Their objective is to minimize the distance between themselves and the top of the mountain. At some point they reach a location where there are several possible paths leading up. They do not know which, if any, eventually lead to the top. A possible technique for solving the climbers' dilemma is to mark this location and strike out on one of the paths. If the chosen path turns out to be unpassable, they can return to the marked location and explore another route. Along one of these paths suppose the climbers come upon another such dilemma. They might mark this location also and take one of the possible paths. The climbers now have two locations to remember. If none of the paths from the second location lead to the top, they can retreat to the lower marked location and choose another path.

The algorithm presented in this dissertation works roughly along the same lines as the search technique employed by the climbers. The locations where the paths split will be called **bases**. At any time there will be m bases, where m is the number of GUB–sets in the problem. If fewer than m dilemmas have been encountered, the larger numbered bases will contain copies of the highest location currently remembered. This is redundant information, but is convenient to handle this way. The bases will be numbered one through m. The base numbered one will be referred to as a 1–base and will represent the location of the first path split, that lowest on the mountain. The 2–base is the location of the next path split. Where this second path split takes place depends on which path was chosen at the first path split, the 1–base. As a result, whenever the climbers retreat to the 1–base to strike out on a new path, they can forget the locations stored as the 2–base, 3–base, etc., that occurred on the other path.

At any point of the algorithm there is a one–one onto mapping from the GUB–sets to the integers $\{1, 2, \ldots, m\}$. The mapping is determined as follows. Let u be the position currently being considered by the algorithm. Then u defines exactly m variables in the problem, one from each GUB–set, that are currently set to one. All other variables are currently set to zero. Each of the variables that is set to one is associated with a "cost" c_{ij}. The GUB–set containing the variable associated with the lowest of these costs is mapped to one. The GUB–set containing the variable with the highest of these costs is mapped to m. All other GUB–sets are mapped to the integer corresponding to their ranking, second lowest mapped to two, etc.

Since some costs may be equal, a consistent tie–breaking procedure is needed. This algorithm uses a least–index rule. For example, if the costs associated with the variables having value 1 in GUB–sets i and j are tied for the lowest cost and $i < j$, GUB–set i is assigned the number one. Many procedures could be used to break ties though some care need be taken in choosing a rule. Requirements for the tie–breaking procedure and proof that the least–index rule satisfies them will be included in Chapter 5.

When the climbers feel their present path will not lead to the top, they will wish to backtrack to the nearest marked location below them and embark on a different path. The analogous action in this algorithm will be called a **step**. There are different types of steps. If the climbers are currently remembering three locations where paths split below them and they wish to retreat to the one closest, this will be referred to as a 3–step. If they decide their error came all the way back at the first location and wish to retreat all the way there to choose a new path, this will be called a 1–step. Note that a j–step involves retreating to the location referred to as the j–base. Since there are m bases that the algorithm will be remembering at any one time, there are m types of steps.

In more mathematical terms, the locations stored in the bases are positions that are defined by position vectors such as u. When the algorithm wants to move further toward a complete solution, from one level of path splitting to the next, the present position is stored as the appropriately numbered base. A step involves backtracking to the base having the same number as the type of the step and incrementing the position of the one in the GUB–set mapped to the number of that base. These numbers associated with each GUB–set, each base and each type of step are called their **order**. It is equivalent to refer to a j–base or a base of order j.

- **Cost Normalization**

Assume a MCIP with objective function

$$\text{minimize} \quad \sum_{i=1}^{m} \sum_{j=1}^{n_i} c_{ij} x_{ij}.$$

has just been solved. This is equivalent to

$$\text{minimize} \quad \sum_{i=1}^{m} c_{iu_i}$$

over all possible position vectors u. If u^* is an optimal position, the optimal objective value is $\sum_{i=1}^{m} c_{iu_i^*}$. Every feasible solution to the problem includes exactly one element from each GUB–set. Hence, if every cost associated with one particular GUB–set were altered by adding some constant K, the optimal objective value would change by precisely K. The set of optimal solutions would remain the same.

Further, if K_i were added to each cost in GUB–set i, for $i = 1, 2, \ldots, m$, the set of optimal solutions would remain the same and the new optimal objective value would be

$$\sum_{i=1}^{m} c_{iu_i^*} + \sum_{i=1}^{m} K_i.$$

An analogous observation can be made for both the rows and columns in the classical assignment problem. This is the basis for Kuhn's Hungarian algorithm.

The importance of this fact to the algorithm at hand is that by adding a sufficiently small (large negative) constant to some GUB–set it can be assured that that GUB–set always has the lowest cost in any base. By adding appropriate constants to each GUB–set , it can be assured that GUB–set 1 is always lowest cost, GUB–set 2 always next, ..., GUB–set m always highest, or any other order chosen. When this is done the algorithm acts as if there is no flexibility in choosing separations. It always separates first by GUB–set 1, then by GUB–set 2, etc. In practice, this has had a negative effect on solution times. With the flexibility to change separations, the algorithm tends to move more directly toward the optimal solution.

The question then arises, if minimum flexibility worsens performance, could some set of constants be added to the costs to increase flexibility and speed up the process? The lowest cost corresponding to each GUB–set could be subtracted from each cost for that GUB–set. Then the revised lowest cost in every GUB–set is zero. The first column becomes completely degenerate. Computationally, this has had little effect on most of the test problems run since the original lowest costs for each GUB–set were very similar. However, on problems where this is not the case, this modification could significantly reduce run times.

The row normalization accomplishes one other important task. It makes all costs non–negative. In the basic statement of the algorithm this is not necessary, but in Chapter 6 an LP based surrogate constraint package will be added that does require this. The traditional method of achieving non–negativity, replacing each variable having a negative cost by its complement, does not work for this problem. It destroys the GUB structure. For example, if there is a GUB–constraint $x_1 + x_2 + x_3 = 1$ with cost $c_2 < 0$, substituting $x'_2 = 1 - x_2$ in the GUB constraint gives $x_1 - x'_2 + x_3 = 0$. This is no longer a GUB constraint.

- **Notation**

Z: The positive integers.

m: The number of GUB–sets in the problem.

n_i: The number of elements in the i^{th} GUB–set, $n_i \in Z$.

n: The number of variables in the problem. $n = \sum_{i=1}^{m} n_i$.

κ: The number of rows in $b + Ax \geq 0$.

u: A position, $u \in Z^m$, $u_i \in \{1, 2, \ldots, n_i\}$.

b^j: j–base. $b^j \in Z^m$, $b_i^j \in \{1, 2, \ldots, n_i\}$.

S: Current partial solution. This is a set containing the indicies of the GUB–sets currently fixed.

$r_i(u)$: The index of the GUB–set corresponding to the i^{th} smallest cost associated with the position u. $r_i(u) \in \{1, 2, \ldots, m\}$.

$K(u)$: The vector of costs associated with the vector u. $K(u) \in \Re^m$.

$\overline{K}(u)$: The total cost associated with the position u. $\overline{K}(u) = \sum_{i=1}^{m} K_i(u)$. $\overline{K}(u) \in \Re$.

$R^j(u)$: The set of indicies with the j lowest associated costs.

$$R^j(u) = \{ r_1(u), r_2(u), \ldots, r_j(u) \}.$$

- **Statement of the Algorithm**

Following is a formal statement of the algorithm. The details of sub-problem solution by inspection will be left to Chapter 6.

Step 0:(Initialization) Order elements in each GUB–set by increasing cost. Normalize the costs associated with each GUB–set. Set $\bar{z} = 1+\sum_{i=1}^{m} c_{n_i}$. This is higher than the cost associated with any feasible solution to the problem. Set $S = \emptyset$ and $j = 0$.

Step 1:(Inspection) If $\overline{K}(b^j) \geq \bar{z}$ go to step 4; if $\overline{K}(b^j) < \bar{z}$ and b^j is feasible, record b^j as the new incumbent and go to step 4. Otherwise, go to step 2.

Step 2:(Further Inspection) If there is no feasible completion of S with value better than $\bar{z}$, go to step 5. If S has an obvious optimal completion with objective value less than $\bar{z}$, record this as the new incumbent and go to step 5. Otherwise, go to step 3.

Step 3:(Augmentation) If $j = m$, go to step 5. Otherwise augment S by $r_j(b^j)$, increment j by 1 and go to step 1.

Step 4:(Backtracking) If $j = 1$ terminate. Otherwise, decrement j by 1 and go to step 5.

Step 5:(Stepping) j–step to a new position u. Set bases of orders $j, j+1, \ldots, m$ to u. Go to step 1.

Figure 3–3 shows a flowchart of this algorithm.

- **Extensions**

It is not difficult to extend this formulation in several directions. For instance, it may be desirable to allow choosing at most one element (rather than exactly one) from each GUB–set, that is, to allow constraints such as $\sum_{j=1}^{n_i} x_{ij} \leq 1$. This can be accomplished by adding a slack 0—1 variable to such GUB–sets.The cost coefficients and constraint coefficients of the slacks are zero. This extension has little cost in run time. As will be seen in Chapter 7, adding variables to existing GUB–sets increases the worst case bound polynomially.

It is possible to extend this formulation to handle constraints such as

$$\sum_{i=1}^{m} x_{ij} = 2.$$

Create a clone GUB–set having the same costs as the original. If x_{ij} and x'_{ij} are the two copies of some variable, in all constraints replace x_{ij} with $x_{ij} + x'_{ij}$. It is then necessary to append an additional general constraint to assure x_{ij} is not chosen from both copies of the GUB–set. One such set of constraints would be

$$x_{ij} + x'_{ij} \leq 1 \quad , i = 1, 2, \ldots, m; \quad j = 1, 2, \ldots, n_i.$$

This extension would have significant negative impact on the performance of the algorithm since a new GUB–set and many constraints are introduced.

A third possible extension is overlapping of the GUB–sets. For instance, x_{ij} may be in GUB–sets i and i'. This can be handled by placing a clone variable x'_{ij} in one of the GUB–sets and then adding the general constraint $x_{ij} = x'_{ij}$. The clone variable would have zero coefficients in cost row and general constraints.

It is also possible to relax the requirement that the GUB–constraints be exhaustive, that is, to allow variables that are not in any GUB–set. A false GUB–set can be created for each variable of this type. Let x_i be some variable in the problem that is not in any GUB–set. Let $x'_i = 1 - x_i$. Then x_i and x'_i form a GUB–set of their own with GUB–constraint $x_i + x'_i = 1$. Assign x'_i a cost of zero and zero coefficients in all general constraints. The GUB structure is now exhaustive and the algorithm can be applied. This procedure adds one GUB–set and on variable to the problem for each original variable not included in a GUB–set.

This extension leads to the realization that any binary program could be formulated so that this algorithm would apply. The efficiency of this algorithm, relative to the additive algorithm, is a function of how extensive the GUB structure is. As will be seen in Chapter 7, if a problem having no GUB structure were formulated for this algorithm, the number of branches that it needs to enumerate would be identical to the number enumerated by a general algorithm. As GUB structure is introduced, the number of branches needed by this algorithm decreases. This is not meant to imply that this algorithm is suggested for solution of general binary programs. Algorithms not designed specially for MCIP can employ techniques that are precluded from use in this algorithm by the exploitation of the special GUB structure.

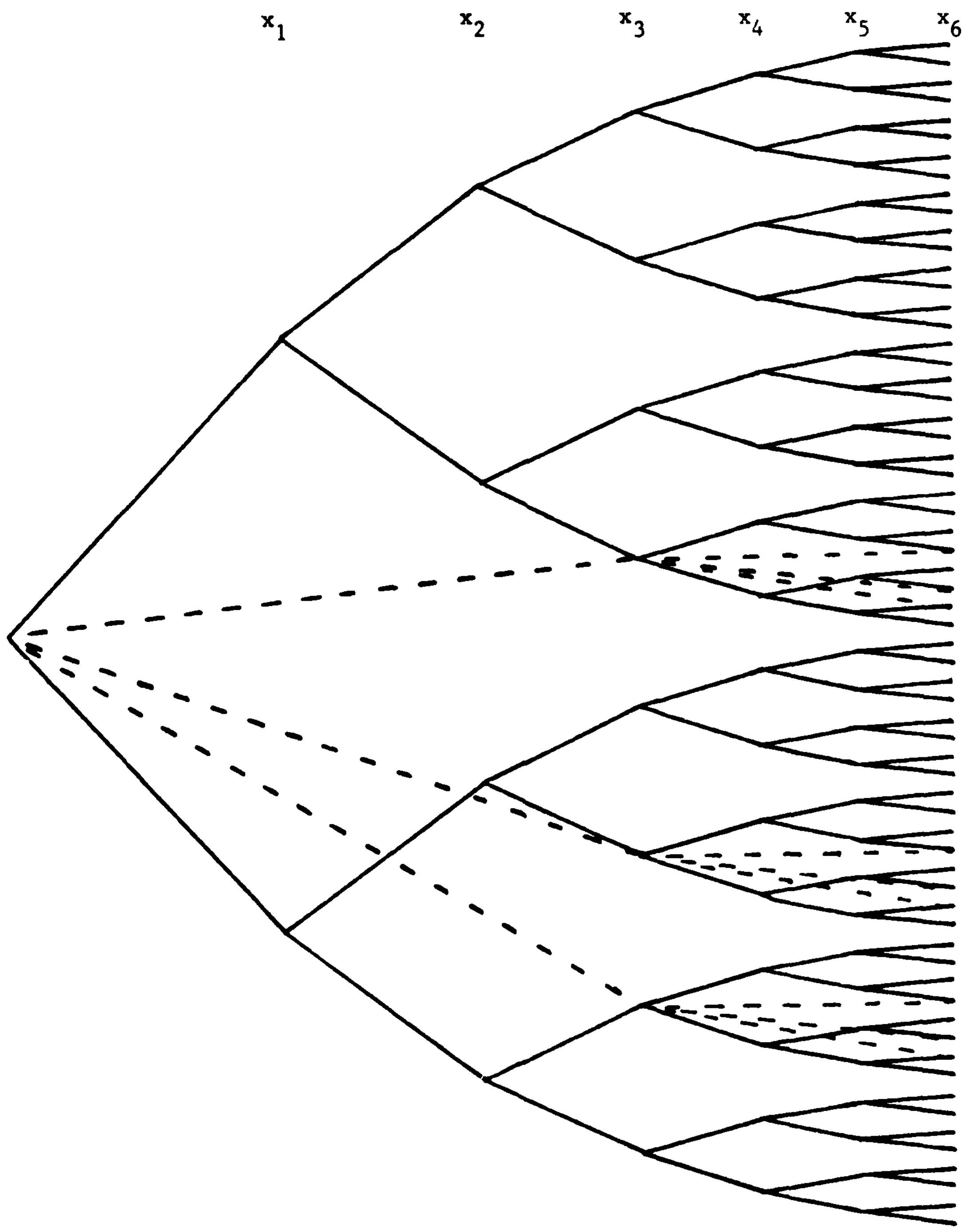


FIGURE 3–1: Tree for a 6 variable binary program.

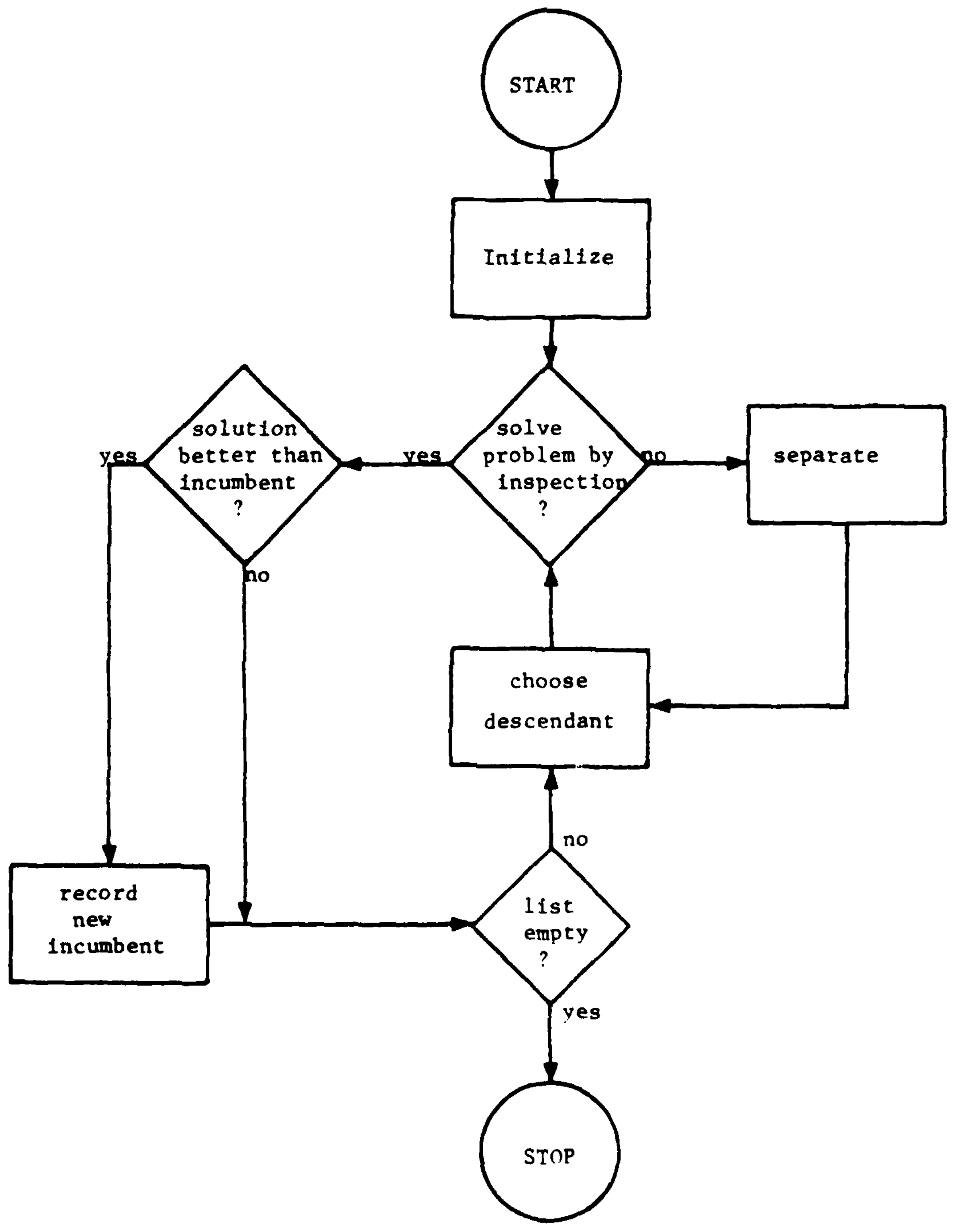


FIGURE 3-2: Flowchart for a general branch-and-bound procedure.

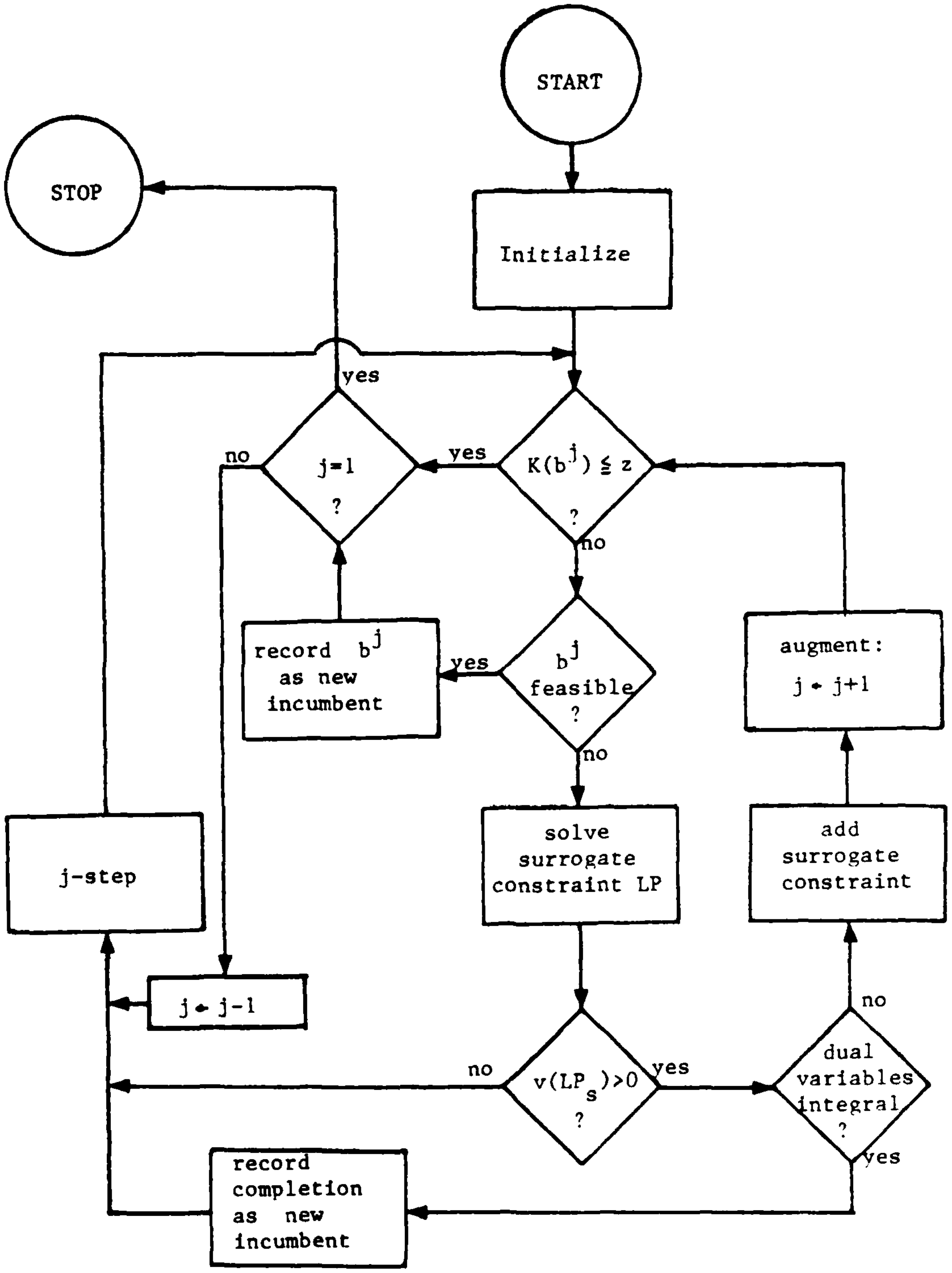

START
STOP
Initialize
yes
no
j=1 ?
yes
$K(b^j) \leq z$?
no
record b^j as new incumbent
yes
b^j feasible ?
no
augment: $j \leftarrow j+1$
j-step
solve surrogate constraint LP
add surrogate constraint
$j \leftarrow j-1$
no
no
$v(LP_s)>0$?
yes
dual variables integral ?
yes
record completion as new incumbent

FIGURE 3-3: Flowchart for the MCIP algorithm.

CHAPTER 4

APPLICATIONS AND EXAMPLES

In Chapter 2 the classical assignment problem, the generalized assignment problem and the multiple choice knapsack problem were seen to be special cases of the multiple choice integer program. Other classes of well studied problems, such as the travelling salesman problem, are also special cases of the MCIP. Each of these subclasses of the MCIP have additional special structure that is exploited in algorithms designed especially for them and not exploited in algorithms designed for the MCIP. As a result there is little hope that any algorithm designed for the MCIP could solve these problems more efficiently than their specially designed algorithms. However, few practitioners of integer programming have at their disposal special codes for each problem they may encounter. There is a place for reasonably efficient algorithms that encompass several of these special smaller classes. This is one use of the algorithm presented in this dissertation.

There are also many problems that are more general than any well studied special case of the MCIP, but that still fit into the formulation of the MCIP. The literature contains many real problems that fit this description or, with little added complexity, quickly become too general for their special cases. Examples include many types of scheduling ([Baker, 1971–1], [Conway, Maxwell & Miller, 1967–1]), selling television time ([Brown, 1969–1]), menu planning ([Sinha & Zoltners, 1979–1]), catalogue space planning ([Johnson, Zoltners & Sinha, 1979–1]), school time tabling ([Lawrie, 1969–1]), facilities location ([Ross & Soland, 1977–1]), assembly line balancing ([Thangavelu & Shetty, 1971–1]), project selection ([Westley, 1980–1]) and many deterministic dynamic programs.

Three examples will be discussed in this chapter to demonstrate how problems can be formulated as MCIP's and how the algorithm presented in Chapter 3 solves them. The version of the algorithm used in this chapter will be only a skeletal version of that used in the code analyzed in Chapter 7. The purpose here is instructional rather than to show efficiency of solution.

- **Scheduling**

The particular scheduling problem that will be demonstrated is athletic team scheduling. The algorithm presented in Chapter 3 was developed as the outgrowth of the analysis of this problem [Bean & Birge, 1980–1].

Consider some point (perhaps the beginning) in the schedule of *m* teams in an athletic league where the location of the teams are known. The problem is to assign the teams to new locations so that another set of games can be played. For each team it is known how far it is from their present location to each of the m cities in the league (including the present location). If one particular team is not to stay in its present location, or if it is not allowed to go to another certain city, the cost involved could be made infinite. Let c_{ij} be the cost for team i to travel from its present location to city j. One determinant of this cost is the distance between the cities. The objective is to minimize the total travel cost for all the teams in moving to new locations. (Note that a new location could be the present location.)

The constraints on the problem require that if a team is to visit another city, the home team of that city must also be assigned there. Formally the problem is

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m}\sum_{j=1}^{m} c_{ij}x_{ij} \\ \text{subject to:} \quad & x_{jj} - \sum_{i \neq j} x_{ij} = 0 \quad , \forall j \\ & \sum_{j=1}^{m} x_{ij} = 1 \quad , \forall i \\ & x_{ij} \in \{0,1\} \quad , \forall i; \forall j, \end{aligned} \qquad (SCH)$$

where $x_{ij} = 1$ if team i is assigned to city j, and 0 otherwise.

The costs in the following example were chosen so that the variables would not need to be reordered in any GUB-set. This should make the problem easier to follow. The example will include four teams. Let the cost matrix be

$$\begin{bmatrix} 0 & 1 & 3 & 6 \\ 1 & 3 & 3 & 7 \\ 3 & 5 & 6 & 9 \\ 2 & 2 & 4 & 5 \end{bmatrix}$$

where each row corresponds to a team and each column to a city. Figure 4–1 depicts the sequence of positions considered by the algorithm in solving this problem. Each matrix in figure 4–2 presents all the bases stored at each iteration of the algorithm. For example, if the base matrix is

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 3 \\ 1 & 1 & 1 & 2 \\ 1 & 1 & 1 & 1 \end{bmatrix},$$

then the 1–base is $(1,1,1,1)^t$, the 2–base is $(1,2,1,1)^t$, the 3–base is $(1,3,1,1)^t$ and the 4–base is $(1,3,2,1)^t$. Just above each matrix in figure 4–1 is the reference number of the position represented in that matrix. Just below each matrix is a comment indicating what characteristic of this position leads the algorithm to its next action. There are three types of comments: "cost" means the position cost fathomed in step 1 of the algorithm, "*l*,inf" means after *l* augmentations step 2 in the algorithm found the partial solution to have no feasible completions, and "inc,*c*" means the position is recorded as the new incumbent with a cost of *c*.

The starting position is $(1,1,1,1)^t$, the lowest cost element in each GUB–set. The algorithm augments GUB–set 1, assigning team 1 to city 1 which is acceptable, then augments GUB–set 2, assigning team 2 to city 1 also. This is also acceptable so that there is now a game at city 1. Next it augments

GUB–set 4 (since the costs are $2 < 3$). This is not acceptable because there are now three teams at city 1. Step 2 of the algorithm states that if there is no feasible completion to the partial solution, which is the case here, the algorithm should go to step 5 which takes a j–step. Note that $j = 3$ now so that the algorithm 3–steps to position 2 of figure 4–1. This changes bases 3 and 4 as can be seen in figure 4–2, base matrix 2.

Still at order 3, the partial position is (1,1,*,2). This has no feasible completion because if team 2 is at city 1, team 4 cannot be at city 2. Hence, step 3 of the algorithm indicates that another 3–step is appropriate. Now at position 3 in figure 4–1, the partial position is (1,1,1,*). Note that the last 3–step caused the costs associated with GUB–sets 3 and 4 to change order. This is an example of choosing a descendant of a different separation as discussed in Chapter 3.

This partial solution has no feasible completion since there are three teams at city 1. Hence, step 2 directs that another 3–step should be taken. But, this 3–step affects GUB–set 3 rather than GUB–set 4 due to the change in cost order. The resulting position is that labelled 4 in figure 4–1.

The partial position is (1,1,*,3) which is acceptable so step 3 of the algorithm augments the remaining GUB–set. The partial position (now a complete position) is (1,1,2,3). This is not feasible so a 4–step is warranted, leading to position 5 in figure 4–1 and a partial position of (1,1,3,3). There is now a game at city 1 between teams 1 and 2, and a game at city 3 between teams 3 and 4. This is feasible, has a cost of $0 + 1 + 6 + 4 = 11$, and is recorded as the new incumbent in step 2 of the algorithm.

The algorithm proceeds to step 4 where j is decremented to 3 and then the algorithm proceeds to step 5 where a 3–step is executed leading to position 6 in figure 4–1. Here the total cost, $\overline{K}(u)$, is 11, which is not less than the incumbent value. Step 1 of the algorithm directs it to step 4 where j is decremented to 2 and then goes to step 5 for a 2–step to position 7 in figure 4–1.

The partial position is now (1,*,*,1), which is acceptable so step 3 of the algorithm augments twice to a partial position (1,2,1,1), which is not feasible (3 teams at city 1). Next the algorithm 4–steps to position 8 in figure 4–1 which is feasible with cost 10. This position is recorded as the new incumbent, j decremented and a 3–step taken.

Next is a 4–step, which leads to position 10 in figure 4–1 and has a cost of 10. This cost fathoms leading to a 3–step to position 11 which also cost fathoms. Next is a 2–step to position 12 which is feasible with cost 8. This position becomes the new incumbent. Then comes a 1–step to postion 13, a 2–step to position 14 and a 1–step to position 15. The cost here is 9 so step 1 of the algorithm directs it to step 4. Since $j = 1$, the algorithm terminates. An optimal solution is team 3 visiting team 1 and team 4 visiting team 2.

As a binary integer program this 16 variable problem has $2^{16} = 65536$ branches to enumerate. As a MCIP it has $4^4 = 256$ branches, of which 15 are enumerated explicitly in this example version of the algorithm. The computer implemented version of the algorithm (see Chapter 7) which uses a sophisticated LP package, investigated zero of the positions to find a correct optimal solution. That is, it found an optimal solution on the first inspection in step 2 of the algorithm with $j = 0$.

- **Assembly Line Balancing**

The assembly line balancing problem is concerned with assigning tasks to stations of an assembly line. The object is to make the total time it takes to complete the tasks at each station as uniform, across the stations, as is possible. For example, if there were four stations and 60 minutes of tasks to complete, it would be desirable to have each station contain tasks that take a total of fifteen minutes. In this case the line can advance one station each fifteen minutes. If the four stations took, respectively, 10 minutes, 20 minutes, 15 minutes and 15 minutes to complete their tasks, stations one, three and four would have to wait on station two before each advance.

The assembly line can be designed to advance at a certain rate, the "production rate." Let the desired production rate be one advancement every γ minutes. That is, it is desired that the assembly line output a product every γ minutes. To accomplish this, every station must have total time not greater than γ. Under this constraint, maximizing uniformity of times is equivalent to minimizing the number of stations, given that the total time for each station is not greater than this maximum time, γ ([Thangavelu & Shetty, 1971–1]). This objective can be formulated in the following manner. Determine a lower bound on the number of stations, n^0 and $n^0 = \lceil T/\gamma \rceil$, where T is the total time of all tasks and $\lceil y \rceil$ is the smallest integer not less than y. For any task assigned to stations $1, 2, \ldots, n^0$ record a penalty of zero. Let m be the total number of tasks. Let p_j be the penalty for assigning a task to position j. Then, recursively, let $p_j = mp_{j-1} + 1$ for $j > n^0$. Now, for $j \geq n^0$, assigning one task to station $j+1$ incurs more penalty than assigning all tasks to station j. Using these penalties as costs, minimizing total cost will minimize the number of stations used.

This problem includes constraints on the total time assignable to any station and also precedence contraints. The tasks may not be assigned in any order, but must conform to a sequence of precedence relationships. For example, suppose that it is necessary for task two to be completed before task 3 can begin. If task two has been completed in a previous station, or if it can be done earlier in the same station, there is no problem. However, task two must not be assigned to a later station than task 3.

These precedence relationships are often represented by a precedence

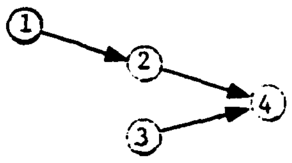


FIGURE 4–3: Precedence diagram.

diagram. If there are four tasks ($m = 4$), one possible precedence diagram is seen in figure 4–3. In this particular diagram, task one must precede task two and tasks two and three must both precede task four. Note that to precede, a task must be in an earlier station or the same station.

Formulating the precedence relationships requires one inequality constraint for each precedence. In figure 4–3 there are three precedences, hence, three precedence constraints. To force task i to precede task i', the constraint is

$$\sum_{j=1}^{n} (n-j)(x_{ij} - x_{i'j}) \geq 0,$$

where n is the maximum possible number of stations in the assembly line (see [Thangavelu & Shetty, 1971–1]).

The example that is shown has four tasks. The maximum possible number of stations is four. The penalties for assigning a task to station j are 0,0,1,5, respectively, for $j = 1,2,3,4$. Let $c_{ij} = p_j$. The times required to complete each of the four tasks are, respectively, 3 minutes, 4 minutes, 7 minutes and 4 minutes. The maximum allowable time in any one station is 10 minutes. The precedence diagram is that in figure 4–3. Then the example can be formulated as

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{4}\sum_{j=1}^{4} c_{ij}x_{ij} \\
\text{subject to:} \quad & 10 - 3x_{1j} - 4x_{2j} - 7x_{3j} - 4x_{4j} \geq 0, \quad j = 1,2,3,4 \\
& \sum_{j=1}^{4}(n-j)(x_{1j} - x_{2j}) \geq 0 \\
& \sum_{j=1}^{4}(n-j)(x_{2j} - x_{4j}) \geq 0 \\
& \sum_{j=1}^{4}(n-j)(x_{3j} - x_{4j}) \geq 0 \\
& \sum_{j=1}^{4} x_{ij} = 1 \quad , i = 1,2,3,4 \\
& x_{ij} \in \{0,1\} \quad , \forall i; \forall j.
\end{aligned}
\tag{ALB}
$$

Figure 4–4 shows the positions investigated by the algorithm in solving this problem, where rows correspond to tasks and columns to stations. Figure 4–5 shows the base matricies at each iteration. The algorithm begins at position 1 in figure 4–4. After augmenting three times, the time for station 1 is 14 minutes (where 10 is maximum) so a 3–step takes place to position 2 in figure 4–4. Now the partial position is (1,1,2,*) which, after an augmentation, has total time of 11 minutes so a 4–step occurs. Now the time for station two is 11 minutes. The algorithm takes a 4–step to position 4 which is feasible with penalty 1. This is recorded as the incumbent.

The algorithm follows with a 3–step to position 5 in figure 4–4 which cost fathoms. The following 2–step leads to position 6. After two augmentations the partial position is (1,1,1,1) which has a total of 14 minutes time for station 1. Hence, a 4–step to position 7 is taken. This is a new incumbent with penalty 0. Next is a 3–step leading to position 8 in figure 4–4 which cost fathoms and then the algorithm takes a 2–step to position 9. This cost fathoms as does position 10 and the algorithm terminates. An optimal solution is to assign tasks 1 and 3 to station 1 and tasks 2 and 4 to station 2.

- **Distribution**

The following problem is a text book example for dynamic programming from [Hillier & Lieberman, 1980–1]. It involves distributing five crates of strawberries amongst three stores to maximize profit.

For ease in reading the positions in figure 4–6 assume the costs had been ordered before the variable assignments. Then $x_{ij} = 1$ implies that store i gets $6 - j$ crates, for $j = 1, 2, \ldots, 6$. To obtain the costs shown in figure 4–6, the costs in the problem were multiplied by -1 (because this algorithm naturally minimizes), and then the costs were normalized as described in

Chapter 3.

The problem is then formulated as

$$\begin{aligned}
&\text{minimize} && \sum_{i=1}^{3}\sum_{j=1}^{6} c_{ij}x_{ij} \\
&\text{subject to:} && 5-\sum_{i=1}^{3}\sum_{j=1}^{6}(6-j)x_{ij} \geq 0 && (DP)\\
& && \sum_{j=1}^{6} x_{ij} = 1 \quad ,i=1,2,3 \\
& && x_{ij} \in \{0,1\} \quad ,\forall i;\forall j.
\end{aligned}$$

The positions considered by the algorithm are presented in figure 4–6 and the bases in figure 4–7. Due to the length of the algorithm's attack on this problem, the reader will be left to follow through it alone. Though the "by hand" version of the algorithm takes 71 iterations to solve this problem, the advanced code discussed in Chapter 7 discovers an optimal solution on the first inspection.

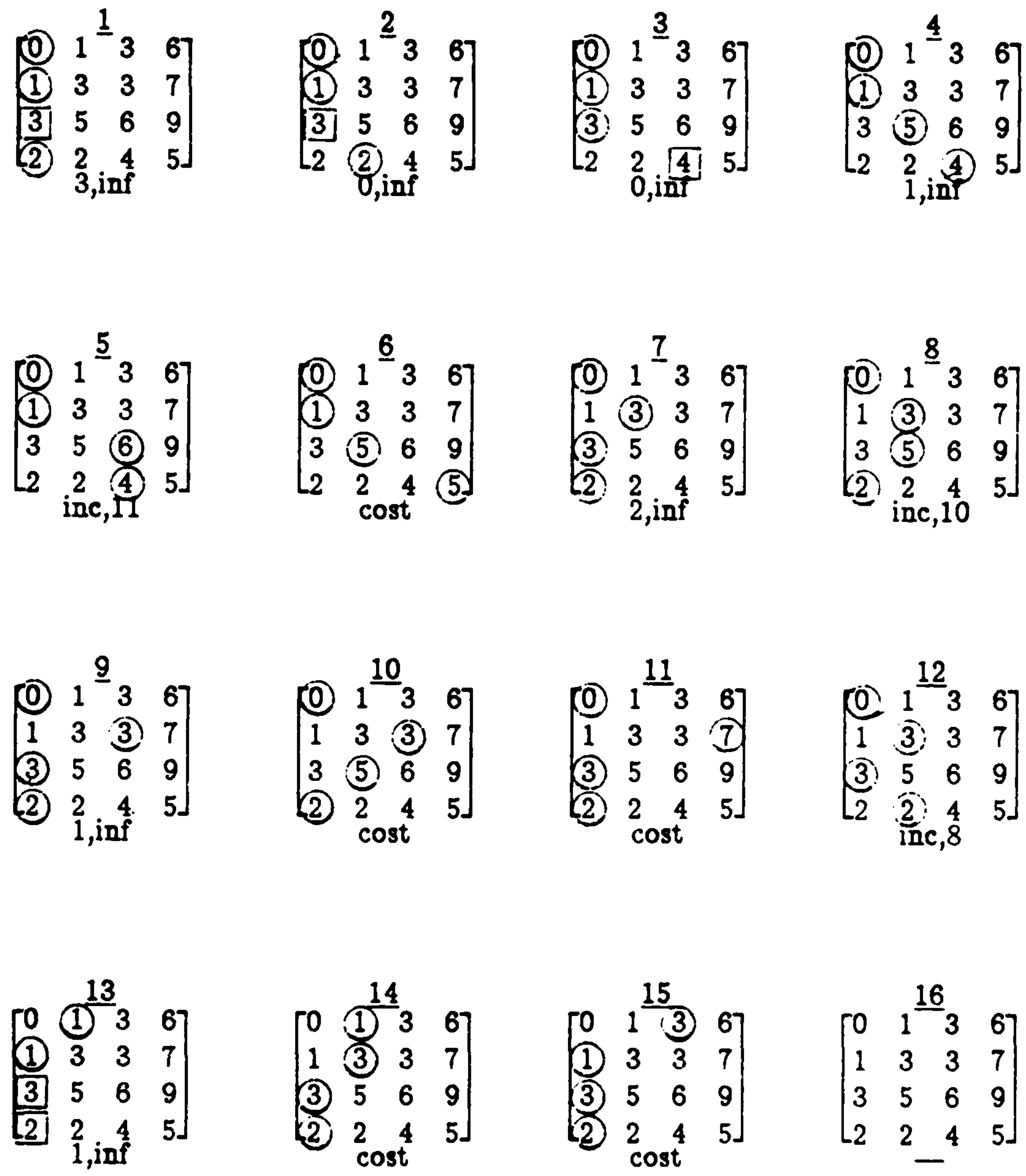


FIGURE 4–1: Positions considered by the scheduling example.

$$\overset{\underline{1}}{\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{bmatrix}}\quad
\overset{\underline{2}}{\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&2&2\end{bmatrix}}\quad
\overset{\underline{3}}{\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&3&3\end{bmatrix}}\quad
\overset{\underline{4}}{\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&2&2\\1&1&3&3\end{bmatrix}}$$

$$\overset{\underline{5}}{\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&2&3\\1&1&3&3\end{bmatrix}}\quad
\overset{\underline{6}}{\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&2&2\\1&1&4&4\end{bmatrix}}\quad
\overset{\underline{7}}{\begin{bmatrix}1&1&1&1\\1&2&2&2\\1&1&1&1\\1&1&1&1\end{bmatrix}}\quad
\overset{\underline{8}}{\begin{bmatrix}1&1&1&1\\1&2&2&2\\1&1&1&2\\1&1&1&1\end{bmatrix}}$$

$$\overset{\underline{9}}{\begin{bmatrix}1&1&1&1\\1&2&3&3\\1&1&1&1\\1&1&1&1\end{bmatrix}}\quad
\overset{\underline{10}}{\begin{bmatrix}1&1&1&1\\1&2&3&3\\1&1&1&2\\1&1&1&1\end{bmatrix}}\quad
\overset{\underline{11}}{\begin{bmatrix}1&1&1&1\\1&2&4&4\\1&1&1&1\\1&1&1&1\end{bmatrix}}\quad
\overset{\underline{12}}{\begin{bmatrix}1&1&1&1\\1&2&2&2\\1&1&1&1\\1&2&2&2\end{bmatrix}}$$

$$\overset{\underline{13}}{\begin{bmatrix}2&2&2&2\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{bmatrix}}\quad
\overset{\underline{14}}{\begin{bmatrix}2&2&2&2\\1&2&2&2\\1&1&1&1\\1&1&1&1\end{bmatrix}}\quad
\overset{\underline{15}}{\begin{bmatrix}3&3&3&3\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{bmatrix}}\quad
\overset{\underline{16}}{\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}}$$

FIGURE 4–2: Bases for the scheduling example.

$$\underset{\text{3,inf}}{\overset{1}{\begin{bmatrix} \textcircled{0} & 0 & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \end{bmatrix}}} \quad \underset{\text{1,inf}}{\overset{2}{\begin{bmatrix} \textcircled{0} & 0 & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \\ 0 & \textcircled{0} & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \end{bmatrix}}} \quad \underset{\text{0,inf}}{\overset{3}{\begin{bmatrix} \textcircled{0} & 0 & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \\ 0 & \textcircled{0} & 1 & 5 \\ 0 & \textcircled{0} & 1 & 5 \end{bmatrix}}} \quad \underset{\text{inc,1}}{\overset{4}{\begin{bmatrix} \textcircled{0} & 0 & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \\ 0 & \textcircled{0} & 1 & 5 \\ 0 & 0 & \textcircled{1} & 5 \end{bmatrix}}}$$

$$\underset{\text{cost}}{\overset{5}{\begin{bmatrix} \textcircled{0} & 0 & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \\ 0 & 0 & \textcircled{1} & 5 \\ \textcircled{0} & 0 & 1 & 5 \end{bmatrix}}} \quad \underset{\text{2,inf}}{\overset{6}{\begin{bmatrix} \textcircled{0} & 0 & 1 & 5 \\ 0 & \textcircled{0} & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \end{bmatrix}}} \quad \underset{\text{inc,0}}{\overset{7}{\begin{bmatrix} \textcircled{0} & 0 & 1 & 5 \\ 0 & \textcircled{0} & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \\ 0 & \textcircled{0} & 1 & 5 \end{bmatrix}}} \quad \underset{\text{cost}}{\overset{8}{\begin{bmatrix} \textcircled{0} & 0 & 1 & 5 \\ 0 & \textcircled{0} & 1 & 5 \\ 0 & \textcircled{0} & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \end{bmatrix}}}$$

$$\underset{\text{cost}}{\overset{9}{\begin{bmatrix} \textcircled{0} & 0 & 1 & 5 \\ 0 & 0 & \textcircled{1} & 5 \\ \textcircled{0} & 0 & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \end{bmatrix}}} \quad \underset{\text{cost}}{\overset{10}{\begin{bmatrix} 0 & \textcircled{0} & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \\ \textcircled{0} & 0 & 1 & 5 \end{bmatrix}}} \quad \underset{\text{—}}{\overset{11}{\begin{bmatrix} 0 & 0 & 1 & 5 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 1 & 5 \end{bmatrix}}} \quad \underset{\text{—}}{\overset{12}{\begin{bmatrix} 0 & 0 & 1 & 5 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 1 & 5 \end{bmatrix}}}$$

FIGURE 4–4: Positions considered during the assembly line balancing example.

$$\overset{\underline{1}}{\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{bmatrix}}\quad\overset{\underline{2}}{\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&2&2\\1&1&1&1\end{bmatrix}}\quad\overset{\underline{3}}{\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&2&2\\1&1&1&2\end{bmatrix}}\quad\overset{\underline{4}}{\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&2&2\\1&1&1&3\end{bmatrix}}$$

$$\overset{\underline{5}}{\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&3&3\\1&1&1&1\end{bmatrix}}\quad\overset{\underline{6}}{\begin{bmatrix}1&1&1&1\\1&2&2&2\\1&1&1&1\\1&1&1&1\end{bmatrix}}\quad\overset{\underline{7}}{\begin{bmatrix}1&1&1&1\\1&2&2&2\\1&1&1&1\\1&1&1&2\end{bmatrix}}\quad\overset{\underline{8}}{\begin{bmatrix}1&1&1&1\\1&2&2&2\\1&1&2&2\\1&1&1&1\end{bmatrix}}$$

$$\overset{\underline{9}}{\begin{bmatrix}1&1&1&1\\1&3&3&3\\1&1&1&1\\1&1&1&1\end{bmatrix}}\quad\overset{\underline{10}}{\begin{bmatrix}2&2&2&2\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{bmatrix}}\quad\overset{\underline{11}}{\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}}\quad\overset{\underline{12}}{\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}}$$

FIGURE 4–5: Bases for the assembly line balancing example.

$$\begin{array}{c} 1 \\ \begin{bmatrix} \textcircled{0} & 1 & 4 & 6 & 10 & 13 \\ \textcircled{0} & 0 & 0 & 1 & 6 & 11 \\ \boxed{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix} \\ 2,\text{inf} \end{array} \qquad \begin{array}{c} 2 \\ \begin{bmatrix} \textcircled{0} & 1 & 4 & 6 & 10 & 13 \\ 0 & \textcircled{0} & 0 & 1 & 6 & 11 \\ \boxed{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \qquad \begin{array}{c} 3 \\ \begin{bmatrix} \textcircled{0} & 1 & 4 & 6 & 10 & 13 \\ 0 & 0 & \textcircled{0} & 1 & 6 & 11 \\ \boxed{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array}$$

$$\begin{array}{c} 4 \\ \begin{bmatrix} \textcircled{0} & 1 & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & \boxed{1} & 6 & 11 \\ \textcircled{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \qquad \begin{array}{c} 5 \\ \begin{bmatrix} \textcircled{0} & 1 & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & \boxed{1} & 6 & 11 \\ 0 & \textcircled{0} & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \qquad \begin{array}{c} 6 \\ \begin{bmatrix} \textcircled{0} & 1 & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & \boxed{1} & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array}$$

$$\begin{array}{c} 7 \\ \begin{bmatrix} \textcircled{0} & 1 & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & 1 & \boxed{6} & 11 \\ 0 & 0 & \textcircled{1} & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \qquad \begin{array}{c} 8 \\ \begin{bmatrix} \textcircled{0} & 1 & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & 1 & \textcircled{6} & 11 \\ 0 & 0 & 1 & \boxed{6} & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \qquad \begin{array}{c} 9 \\ \begin{bmatrix} \textcircled{0} & 1 & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & 1 & 6 & \boxed{11} \\ 0 & 0 & 1 & \textcircled{6} & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array}$$

$$\begin{array}{c} 10 \\ \begin{bmatrix} \textcircled{0} & 1 & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & 1 & 6 & \boxed{11} \\ 0 & 0 & 1 & 6 & \textcircled{8} & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \qquad \begin{array}{c} 11 \\ \begin{bmatrix} \textcircled{0} & 1 & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & 1 & 6 & \textcircled{11} \\ 0 & 0 & 1 & 6 & 8 & \textcircled{12} \end{bmatrix} \\ \text{inc},23 \end{array} \qquad \begin{array}{c} 12 \\ \begin{bmatrix} 0 & \boxed{1} & 4 & 6 & 10 & 13 \\ \textcircled{0} & 0 & 0 & 1 & 6 & 11 \\ \textcircled{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix} \\ 1,\text{inf} \end{array}$$

$$\begin{array}{c} 13 \\ \begin{bmatrix} 0 & \boxed{1} & 4 & 6 & 10 & 13 \\ \textcircled{0} & 0 & 0 & 1 & 6 & 11 \\ 0 & \textcircled{0} & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \qquad \begin{array}{c} 14 \\ \begin{bmatrix} 0 & \textcircled{1} & 4 & 6 & 10 & 13 \\ \textcircled{0} & 0 & 0 & 1 & 6 & 11 \\ 0 & 0 & \boxed{1} & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \qquad \begin{array}{c} 15 \\ \begin{bmatrix} 0 & 1 & \boxed{4} & 6 & 10 & 13 \\ \textcircled{0} & 0 & 0 & 1 & 6 & 11 \\ 0 & 0 & \textcircled{1} & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array}$$

$$\begin{array}{c} 16 \\ \begin{bmatrix} 0 & 1 & \textcircled{4} & 6 & 10 & 13 \\ \textcircled{0} & 0 & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & \boxed{6} & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \qquad \begin{array}{c} 17 \\ \begin{bmatrix} 0 & 1 & 4 & \textcircled{6} & 10 & 13 \\ \textcircled{0} & 0 & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & \boxed{6} & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \qquad \begin{array}{c} 18 \\ \begin{bmatrix} 0 & 1 & 4 & 6 & \boxed{10} & 13 \\ \textcircled{0} & 0 & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & \textcircled{6} & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array}$$

$$\underset{\text{0,inf}}{\overset{\underline{19}}{\begin{bmatrix} 0 & 1 & 4 & 6 & \boxed{10} & 13 \\ \textcircled{0} & 0 & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & 6 & \textcircled{8} & 12 \end{bmatrix}}} \quad \underset{\text{0,inf}}{\overset{\underline{20}}{\begin{bmatrix} 0 & 1 & 4 & 6 & \textcircled{10} & 13 \\ \textcircled{0} & 0 & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & 6 & 8 & \boxed{12} \end{bmatrix}}} \quad \underset{\text{cost}}{\overset{\underline{21}}{\begin{bmatrix} 0 & 1 & 4 & 6 & 10 & \textcircled{13} \\ \textcircled{0} & 0 & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & 6 & 8 & \textcircled{12} \end{bmatrix}}}$$

$$\underset{\text{1,inf}}{\overset{\underline{22}}{\begin{bmatrix} 0 & \boxed{1} & 4 & 6 & 10 & 13 \\ 0 & \textcircled{0} & 0 & 1 & 6 & 11 \\ \textcircled{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix}}} \quad \underset{\text{0,inf}}{\overset{\underline{23}}{\begin{bmatrix} 0 & \boxed{1} & 4 & 6 & 10 & 13 \\ 0 & \textcircled{0} & 0 & 1 & 6 & 11 \\ 0 & \textcircled{0} & 1 & 6 & 8 & 12 \end{bmatrix}}} \quad \underset{\text{0,inf}}{\overset{\underline{24}}{\begin{bmatrix} 0 & \textcircled{1} & 4 & 6 & 10 & 13 \\ 0 & \textcircled{0} & 0 & 1 & 6 & 11 \\ 0 & 0 & \boxed{1} & 6 & 8 & 12 \end{bmatrix}}}$$

$$\underset{\text{0,inf}}{\overset{\underline{25}}{\begin{bmatrix} 0 & 1 & \boxed{4} & 6 & 10 & 13 \\ 0 & \textcircled{0} & 0 & 1 & 6 & 11 \\ 0 & 0 & \textcircled{1} & 6 & 8 & 12 \end{bmatrix}}} \quad \underset{\text{0,inf}}{\overset{\underline{26}}{\begin{bmatrix} 0 & 1 & \textcircled{4} & 6 & 10 & 13 \\ 0 & \textcircled{0} & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & \boxed{6} & 8 & 12 \end{bmatrix}}} \quad \underset{\text{0,inf}}{\overset{\underline{27}}{\begin{bmatrix} 0 & 1 & 4 & \textcircled{6} & 10 & 13 \\ 0 & \textcircled{0} & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & \boxed{6} & 8 & 12 \end{bmatrix}}}$$

$$\underset{\text{0,inf}}{\overset{\underline{28}}{\begin{bmatrix} 0 & 1 & 4 & 6 & \boxed{10} & 13 \\ 0 & \textcircled{0} & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & \textcircled{6} & 8 & 12 \end{bmatrix}}} \quad \underset{\text{0,inf}}{\overset{\underline{29}}{\begin{bmatrix} 0 & 1 & 4 & 6 & \boxed{10} & 13 \\ 0 & \textcircled{0} & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & 6 & \textcircled{8} & 12 \end{bmatrix}}} \quad \underset{\text{inc,22}}{\overset{\underline{30}}{\begin{bmatrix} 0 & 1 & 4 & 6 & \textcircled{10} & 13 \\ 0 & \textcircled{0} & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & 6 & 8 & \textcircled{12} \end{bmatrix}}}$$

$$\underset{\text{1,inf}}{\overset{\underline{31}}{\begin{bmatrix} 0 & \boxed{1} & 4 & 6 & 10 & 13 \\ 0 & 0 & \textcircled{0} & 1 & 6 & 11 \\ \textcircled{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix}}} \quad \underset{\text{0,inf}}{\overset{\underline{32}}{\begin{bmatrix} 0 & \boxed{1} & 4 & 6 & 10 & 13 \\ 0 & 0 & \textcircled{0} & 1 & 6 & 11 \\ 0 & \textcircled{0} & 1 & 6 & 8 & 12 \end{bmatrix}}} \quad \underset{\text{0,inf}}{\overset{\underline{33}}{\begin{bmatrix} 0 & \textcircled{1} & 4 & 6 & 10 & 13 \\ 0 & 0 & \textcircled{0} & 1 & 6 & 11 \\ 0 & 0 & \boxed{1} & 6 & 8 & 12 \end{bmatrix}}}$$

$$\underset{\text{0,inf}}{\overset{\underline{34}}{\begin{bmatrix} 0 & 1 & \boxed{4} & 6 & 10 & 13 \\ 0 & 0 & \textcircled{0} & 1 & 6 & 11 \\ 0 & 0 & \textcircled{1} & 6 & 8 & 12 \end{bmatrix}}} \quad \underset{\text{0,inf}}{\overset{\underline{35}}{\begin{bmatrix} 0 & 1 & \textcircled{4} & 6 & 10 & 13 \\ 0 & 0 & \textcircled{0} & 1 & 6 & 11 \\ 0 & 0 & 1 & \boxed{6} & 8 & 12 \end{bmatrix}}} \quad \underset{\text{0,inf}}{\overset{\underline{36}}{\begin{bmatrix} 0 & 1 & 4 & \textcircled{6} & 10 & 13 \\ 0 & 0 & \textcircled{0} & 1 & 6 & 11 \\ 0 & 0 & 1 & \boxed{6} & 8 & 12 \end{bmatrix}}}$$

$$\begin{array}{c} \underline{37} \\ \begin{bmatrix} 0 & 1 & 4 & 6 & \boxed{10} & 13 \\ 0 & 0 & \textcircled{0} & 1 & 6 & 11 \\ 0 & 0 & 1 & \textcircled{6} & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \quad \begin{array}{c} \underline{38} \\ \begin{bmatrix} 0 & 1 & 4 & 6 & \boxed{10} & 13 \\ 0 & 0 & \textcircled{0} & 1 & 6 & 11 \\ 0 & 0 & 1 & 6 & \textcircled{8} & 12 \end{bmatrix} \\ \text{inc},18 \end{array} \quad \begin{array}{c} \underline{39} \\ \begin{bmatrix} 0 & \textcircled{1} & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & \boxed{1} & 6 & 11 \\ \textcircled{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix} \\ 1,\text{inf} \end{array}$$

$$\begin{array}{c} \underline{40} \\ \begin{bmatrix} 0 & 1 & \boxed{4} & 6 & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ \textcircled{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \quad \begin{array}{c} \underline{41} \\ \begin{bmatrix} 0 & 1 & \textcircled{4} & 6 & 10 & 13 \\ 0 & 0 & 0 & 1 & \boxed{6} & 11 \\ \textcircled{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \quad \begin{array}{c} \underline{42} \\ \begin{bmatrix} 0 & 1 & 4 & \textcircled{6} & 10 & 13 \\ 0 & 0 & 0 & 1 & \boxed{6} & 11 \\ \textcircled{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array}$$

$$\begin{array}{c} \underline{43} \\ \begin{bmatrix} 0 & 1 & 4 & 6 & \boxed{10} & 13 \\ 0 & 0 & 0 & 1 & \textcircled{6} & 11 \\ \textcircled{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \quad \begin{array}{c} \underline{44} \\ \begin{bmatrix} 0 & 1 & 4 & 6 & \textcircled{10} & 13 \\ 0 & 0 & 0 & 1 & 6 & \textcircled{11} \\ \textcircled{0} & 0 & 1 & 6 & 8 & 12 \end{bmatrix} \\ \text{cost} \end{array} \quad \begin{array}{c} \underline{45} \\ \begin{bmatrix} 0 & \textcircled{1} & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & \textcircled{0} & 1 & 6 & 8 & 12 \end{bmatrix} \\ 1,\text{inf} \end{array}$$

$$\begin{array}{c} \underline{46} \\ \begin{bmatrix} 0 & 1 & \boxed{4} & 6 & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & \textcircled{0} & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \quad \begin{array}{c} \underline{47} \\ \begin{bmatrix} 0 & 1 & \textcircled{4} & 6 & 10 & 13 \\ 0 & 0 & 0 & 1 & \boxed{6} & 11 \\ 0 & \textcircled{0} & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \quad \begin{array}{c} \underline{48} \\ \begin{bmatrix} 0 & 1 & 4 & \textcircled{6} & 10 & 13 \\ 0 & 0 & 0 & 1 & \boxed{6} & 11 \\ 0 & \textcircled{0} & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array}$$

$$\begin{array}{c} \underline{49} \\ \begin{bmatrix} 0 & 1 & 4 & 6 & \boxed{10} & 13 \\ 0 & 0 & 0 & 1 & \textcircled{6} & 11 \\ 0 & \textcircled{0} & 1 & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \quad \begin{array}{c} \underline{50} \\ \begin{bmatrix} 0 & 1 & 4 & 6 & \textcircled{10} & 13 \\ 0 & 0 & 0 & 1 & 6 & \textcircled{11} \\ 0 & \textcircled{0} & 1 & 6 & 8 & 12 \end{bmatrix} \\ \text{cost} \end{array} \quad \begin{array}{c} \underline{51} \\ \begin{bmatrix} 0 & \textcircled{1} & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & \boxed{1} & 6 & 8 & 12 \end{bmatrix} \\ 1,\text{inf} \end{array}$$

$$\begin{array}{c} \underline{52} \\ \begin{bmatrix} 0 & \textcircled{1} & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & 1 & \boxed{6} & 11 \\ 0 & 0 & \textcircled{1} & 6 & 8 & 12 \end{bmatrix} \\ 0,\text{inf} \end{array} \quad \begin{array}{c} \underline{53} \\ \begin{bmatrix} 0 & \textcircled{1} & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & 1 & \textcircled{6} & 11 \\ 0 & 0 & 1 & \textcircled{6} & 8 & 12 \end{bmatrix} \\ 1,\text{inf} \end{array} \quad \begin{array}{c} \underline{54} \\ \begin{bmatrix} 0 & \textcircled{1} & 4 & 6 & 10 & 13 \\ 0 & 0 & 0 & 1 & \textcircled{6} & 11 \\ 0 & 0 & 1 & 6 & \textcircled{8} & 12 \end{bmatrix} \\ 0,\text{inf} \end{array}$$

$$\begin{bmatrix} 0 & \textcircled{1} & 4 & \overset{\underline{55}}{6} & 10 & 13 \\ 0 & 0 & 0 & 1 & \textcircled{6} & 11 \\ 0 & 0 & 1 & \underset{\text{cost}}{6} & 8 & \textcircled{12} \end{bmatrix} \quad \begin{bmatrix} 0 & \textcircled{1} & 4 & \overset{\underline{56}}{6} & 10 & 13 \\ 0 & 0 & 0 & 1 & 6 & \textcircled{11} \\ 0 & 0 & 1 & \underset{\text{cost}}{\textcircled{6}} & 8 & 12 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & \textcircled{4} & \overset{\underline{57}}{6} & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & \textcircled{1} & \underset{\text{2,inf}}{6} & 8 & 12 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 4 & \overset{\underline{58}}{\textcircled{6}} & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & \textcircled{1} & \underset{\text{0,inf}}{6} & 8 & 12 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 4 & \overset{\underline{59}}{6} & \textcircled{10} & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & \textcircled{1} & \underset{\text{0,inf}}{6} & 8 & 12 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 4 & \overset{\underline{60}}{6} & 10 & \textcircled{13} \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & \textcircled{1} & \underset{\text{inc,15}}{6} & 8 & 12 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & \textcircled{4} & \overset{\underline{61}}{6} & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & 1 & \underset{\text{1,inf}}{\textcircled{6}} & 8 & 12 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & \textcircled{4} & \overset{\underline{62}}{6} & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & 1 & \underset{\text{0,inf}}{6} & \textcircled{8} & 12 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & \textcircled{4} & \overset{\underline{63}}{6} & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & 1 & \underset{\text{cost}}{6} & 8 & \textcircled{12} \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 4 & \overset{\underline{64}}{\textcircled{6}} & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & 1 & \underset{\text{1,inf}}{\textcircled{6}} & 8 & 12 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 4 & \overset{\underline{65}}{\textcircled{6}} & 10 & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & 1 & \underset{\text{cost}}{6} & \textcircled{8} & 12 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 4 & \overset{\underline{66}}{6} & \textcircled{10} & 13 \\ 0 & 0 & 0 & \textcircled{1} & 6 & 11 \\ 0 & 0 & 1 & \underset{\text{cost}}{\textcircled{6}} & 8 & 12 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & \textcircled{4} & \overset{\underline{67}}{6} & 10 & 13 \\ 0 & 0 & 0 & 1 & \textcircled{6} & 11 \\ 0 & 0 & \textcircled{1} & \underset{\text{1,inf}}{6} & 8 & 12 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 4 & \overset{\underline{68}}{\textcircled{6}} & 10 & 13 \\ 0 & 0 & 0 & 1 & \textcircled{6} & 11 \\ 0 & 0 & \textcircled{1} & \underset{\text{1,inf}}{6} & 8 & 12 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 4 & \overset{\underline{69}}{\textcircled{6}} & 10 & 13 \\ 0 & 0 & 0 & 1 & 6 & \textcircled{11} \\ 0 & 0 & \textcircled{1} & \underset{\text{cost}}{6} & 8 & 12 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 4 & \overset{\underline{70}}{6} & \textcircled{10} & 13 \\ 0 & 0 & 0 & 1 & \textcircled{6} & 11 \\ 0 & 0 & \textcircled{1} & \underset{\text{cost}}{6} & 8 & 12 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & \textcircled{4} & \overset{\underline{71}}{6} & 10 & 13 \\ 0 & 0 & 0 & 1 & \textcircled{6} & 11 \\ 0 & 0 & 1 & \underset{\text{cost}}{\textcircled{6}} & 8 & 12 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 4 & \overset{\underline{72}}{6} & 10 & 13 \\ 0 & 0 & 0 & 1 & 6 & 11 \\ 0 & 0 & 1 & \underline{6} & 8 & 12 \end{bmatrix}$$

FIGURE 4–6: Positions considered during the dynamic programming example.

$$\begin{array}{c}\underline{1}\\ \begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{2}\\ \begin{bmatrix}1 & 1 & 1\\ 1 & 2 & 2\\ 1 & 1 & 1\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{3}\\ \begin{bmatrix}1 & 1 & 1\\ 1 & 3 & 3\\ 1 & 1 & 1\end{bmatrix}\end{array}$$

$$\begin{array}{c}\underline{4}\\ \begin{bmatrix}1 & 1 & 1\\ 1 & 4 & 4\\ 1 & 1 & 1\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{5}\\ \begin{bmatrix}1 & 1 & 1\\ 1 & 4 & 4\\ 1 & 2 & 2\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{6}\\ \begin{bmatrix}1 & 1 & 1\\ 1 & 4 & 4\\ 1 & 3 & 3\end{bmatrix}\end{array}$$

$$\begin{array}{c}\underline{7}\\ \begin{bmatrix}1 & 1 & 1\\ 1 & 5 & 5\\ 1 & 3 & 3\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{8}\\ \begin{bmatrix}1 & 1 & 1\\ 1 & 5 & 5\\ 1 & 4 & 4\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{9}\\ \begin{bmatrix}1 & 1 & 1\\ 1 & 6 & 6\\ 1 & 4 & 4\end{bmatrix}\end{array}$$

$$\begin{array}{c}\underline{10}\\ \begin{bmatrix}1 & 1 & 1\\ 1 & 6 & 6\\ 1 & 5 & 5\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{11}\\ \begin{bmatrix}1 & 1 & 1\\ 1 & 6 & 6\\ 1 & 6 & 6\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{12}\\ \begin{bmatrix}2 & 2 & 2\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}\end{array}$$

$$\begin{array}{c}\underline{13}\\ \begin{bmatrix}2 & 2 & 2\\ 1 & 1 & 1\\ 1 & 2 & 2\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{14}\\ \begin{bmatrix}2 & 2 & 2\\ 1 & 1 & 1\\ 1 & 3 & 3\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{15}\\ \begin{bmatrix}2 & 3 & 3\\ 1 & 1 & 1\\ 1 & 3 & 3\end{bmatrix}\end{array}$$

$$\begin{array}{c}\underline{16}\\ \begin{bmatrix}2 & 3 & 3\\ 1 & 1 & 1\\ 1 & 4 & 4\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{17}\\ \begin{bmatrix}2 & 4 & 4\\ 1 & 1 & 1\\ 1 & 4 & 4\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{18}\\ \begin{bmatrix}2 & 5 & 5\\ 1 & 1 & 1\\ 1 & 4 & 4\end{bmatrix}\end{array}$$

$$\overset{\underline{19}}{\begin{bmatrix}2&5&5\\1&1&1\\1&5&5\end{bmatrix}} \qquad \overset{\underline{20}}{\begin{bmatrix}2&5&5\\1&1&1\\1&6&6\end{bmatrix}} \qquad \overset{\underline{21}}{\begin{bmatrix}2&6&6\\1&1&1\\1&6&6\end{bmatrix}}$$

$$\overset{\underline{22}}{\begin{bmatrix}2&2&2\\2&2&2\\1&1&1\end{bmatrix}} \qquad \overset{\underline{23}}{\begin{bmatrix}2&2&2\\2&2&2\\1&2&2\end{bmatrix}} \qquad \overset{\underline{24}}{\begin{bmatrix}2&2&2\\2&2&2\\1&3&3\end{bmatrix}}$$

$$\overset{\underline{25}}{\begin{bmatrix}2&3&3\\2&2&2\\1&3&3\end{bmatrix}} \qquad \overset{\underline{26}}{\begin{bmatrix}2&3&3\\2&2&2\\1&4&4\end{bmatrix}} \qquad \overset{\underline{27}}{\begin{bmatrix}2&4&4\\2&2&2\\1&4&4\end{bmatrix}}$$

$$\overset{\underline{28}}{\begin{bmatrix}2&5&5\\2&2&2\\1&4&4\end{bmatrix}} \qquad \overset{\underline{29}}{\begin{bmatrix}2&5&5\\2&2&2\\1&5&5\end{bmatrix}} \qquad \overset{\underline{30}}{\begin{bmatrix}2&5&5\\2&2&2\\1&6&6\end{bmatrix}}$$

$$\overset{\underline{31}}{\begin{bmatrix}2&2&2\\3&3&3\\1&1&1\end{bmatrix}} \qquad \overset{\underline{32}}{\begin{bmatrix}2&2&2\\3&3&3\\1&2&2\end{bmatrix}} \qquad \overset{\underline{33}}{\begin{bmatrix}2&2&2\\3&3&3\\1&3&3\end{bmatrix}}$$

$$\overset{\underline{34}}{\begin{bmatrix}2&3&3\\3&3&3\\1&3&3\end{bmatrix}} \qquad \overset{\underline{35}}{\begin{bmatrix}2&3&3\\3&3&3\\1&4&4\end{bmatrix}} \qquad \overset{\underline{36}}{\begin{bmatrix}2&4&4\\3&3&3\\1&4&4\end{bmatrix}}$$

$$\overset{\underline{37}}{\begin{bmatrix} 2 & 5 & 5 \\ 3 & 3 & 3 \\ 1 & 4 & 4 \end{bmatrix}} \qquad \overset{\underline{38}}{\begin{bmatrix} 2 & 5 & 5 \\ 3 & 3 & 3 \\ 1 & 5 & 5 \end{bmatrix}} \qquad \overset{\underline{39}}{\begin{bmatrix} 2 & 2 & 2 \\ 4 & 4 & 4 \\ 1 & 1 & 1 \end{bmatrix}}$$

$$\overset{\underline{40}}{\begin{bmatrix} 2 & 3 & 3 \\ 4 & 4 & 4 \\ 1 & 1 & 1 \end{bmatrix}} \qquad \overset{\underline{41}}{\begin{bmatrix} 2 & 3 & 3 \\ 4 & 5 & 5 \\ 1 & 1 & 1 \end{bmatrix}} \qquad \overset{\underline{42}}{\begin{bmatrix} 2 & 4 & 4 \\ 4 & 5 & 5 \\ 1 & 1 & 1 \end{bmatrix}}$$

$$\overset{\underline{43}}{\begin{bmatrix} 2 & 5 & 5 \\ 4 & 5 & 5 \\ 1 & 1 & 1 \end{bmatrix}} \qquad \overset{\underline{44}}{\begin{bmatrix} 2 & 5 & 5 \\ 4 & 6 & 6 \\ 1 & 1 & 1 \end{bmatrix}} \qquad \overset{\underline{45}}{\begin{bmatrix} 2 & 2 & 2 \\ 4 & 4 & 4 \\ 2 & 2 & 2 \end{bmatrix}}$$

$$\overset{\underline{46}}{\begin{bmatrix} 2 & 3 & 3 \\ 4 & 4 & 4 \\ 2 & 2 & 2 \end{bmatrix}} \qquad \overset{\underline{47}}{\begin{bmatrix} 2 & 3 & 3 \\ 4 & 5 & 5 \\ 2 & 2 & 2 \end{bmatrix}} \qquad \overset{\underline{48}}{\begin{bmatrix} 2 & 4 & 4 \\ 4 & 5 & 5 \\ 2 & 2 & 2 \end{bmatrix}}$$

$$\overset{\underline{49}}{\begin{bmatrix} 2 & 5 & 5 \\ 4 & 5 & 5 \\ 2 & 2 & 2 \end{bmatrix}} \qquad \overset{\underline{50}}{\begin{bmatrix} 2 & 5 & 5 \\ 4 & 6 & 6 \\ 2 & 2 & 2 \end{bmatrix}} \qquad \overset{\underline{51}}{\begin{bmatrix} 2 & 2 & 2 \\ 4 & 4 & 4 \\ 3 & 3 & 3 \end{bmatrix}}$$

$$\overset{\underline{52}}{\begin{bmatrix} 2 & 2 & 2 \\ 4 & 5 & 5 \\ 3 & 3 & 3 \end{bmatrix}} \qquad \overset{\underline{53}}{\begin{bmatrix} 2 & 2 & 2 \\ 4 & 5 & 5 \\ 3 & 4 & 4 \end{bmatrix}} \qquad \overset{\underline{54}}{\begin{bmatrix} 2 & 2 & 2 \\ 4 & 5 & 5 \\ 3 & 4 & 5 \end{bmatrix}}$$

$$\begin{array}{c}\underline{55}\\ \begin{bmatrix}2&2&2\\4&5&5\\3&4&6\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{56}\\ \begin{bmatrix}2&2&2\\4&6&6\\3&4&4\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{57}\\ \begin{bmatrix}3&3&3\\4&4&4\\3&3&3\end{bmatrix}\end{array}$$

$$\begin{array}{c}\underline{58}\\ \begin{bmatrix}3&3&4\\4&4&4\\3&3&3\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{59}\\ \begin{bmatrix}3&3&5\\4&4&4\\3&3&3\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{60}\\ \begin{bmatrix}3&3&6\\4&4&4\\3&3&3\end{bmatrix}\end{array}$$

$$\begin{array}{c}\underline{61}\\ \begin{bmatrix}3&3&3\\4&4&4\\3&4&4\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{62}\\ \begin{bmatrix}3&3&3\\4&4&4\\3&4&5\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{63}\\ \begin{bmatrix}3&3&3\\4&4&4\\3&4&6\end{bmatrix}\end{array}$$

$$\begin{array}{c}\underline{64}\\ \begin{bmatrix}3&4&4\\4&4&4\\3&4&4\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{65}\\ \begin{bmatrix}3&4&4\\4&4&4\\3&4&5\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{66}\\ \begin{bmatrix}3&5&5\\4&4&4\\3&4&4\end{bmatrix}\end{array}$$

$$\begin{array}{c}\underline{67}\\ \begin{bmatrix}3&3&3\\5&5&5\\3&3&3\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{68}\\ \begin{bmatrix}3&4&4\\5&5&5\\3&3&3\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{69}\\ \begin{bmatrix}3&4&4\\5&5&6\\3&3&3\end{bmatrix}\end{array}$$

$$\begin{array}{c}\underline{70}\\ \begin{bmatrix}3&5&5\\5&5&5\\3&3&3\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{71}\\ \begin{bmatrix}3&3&3\\5&5&5\\4&4&4\end{bmatrix}\end{array} \qquad \begin{array}{c}\underline{72}\\ \begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}\end{array}$$

FIGURE 4–7: Bases for the dynamic programming example.

CHAPTER 5

PROOF OF CONVERGENCE OF THE ALGORITHM

The algorithm described in Chapter 3 finds an optimal solution to MCIP, or returns an infeasibility condition, in finite time. This fact shall be verified by the following line of reasoning. Consider the algorithm with all fathoms eliminated. After demonstrating that no position can be explicitly considered more than once, and using the fact that there are only a finite number of positions, it can be concluded that the algorithm in this form terminates in a finite amount of time. By then showing that this form of the algorithm considers every position, it can be concluded that it finds an optimal solution or returns an infeasibility condition. Finally, it is argued that introducing the cost and feasibility fathoms does not cause the algorithm to cycle or all optimal solutions to be skipped. Hence, the algorithm in its full version finds an optimal solution or returns an infeasibility condition.

Several lemmas will be presented to aid in proving the major theorems. First it is argued that, between consecutive j–steps, the algorithm considers a subproblem including a subset of $m - j$ GUB–sets from the original problem. The other j are fixed and will not change before the next j–step. The analogous characteristic in general implicit enumeration is that, given a partial solution, S, the problem can be treated as an implicit enumeration problem with $n - |S|$ variables until some variable in S is freed or changed (where $|S|$ denotes the cardinality of S). It is clear in the general case that this decomposition is valid. In the case of this algorithm it is not so obvious. Difficulty arises because of the continual switching in the order associated with each GUB–set.

In the following lemmas it will be assumed that the sequence of bases generated by the algorithm is given, without reference to the types of fathoming allowed. The sequence of positions generated by the algorithm with any fathoming scheme is a subsequence of that generated by the algorithm without any fathoming. Allowing stronger fathoming criteria simply permits more of the positions to be skipped, that is, implicitly enumerated. As a result, the following lemmas will hold for any fathoming policy.

Lemma 5–1: At any time during the algorithm and any $1 \le j \le i \le m$, $b^i_k = b^j_k$ for $k \in R^j(b^j) = R^j(b^i)$.

Proof: At the time of the last step of order j or lower, b^j and b^i were set to be equal. Since then, only steps of order $(j+1)$ through m can have occurred. These steps change only the $(j+1)^{st}$ through m^{th} smallest costs and, hence, have no effect of the j lowest cost elements of b^j, those in $R^j(b^j)$. By lemma 5–5, those steps of order $(j+1)$ through m can not have decreased costs associated with their GUB–sets. By lemma 5–6, if costs were equal, the j lowest ordered GUB–sets are still lowest ordered. Hence, the j lowest cost elements before these steps are still lowest and $R^j(b^j) = R^j(b^i)$.■

It is clear that 1–bases are non–decreasing during the process of the algorithm. The only time that bases of any order are decreased is when a step of lower order is performed. Since there are no steps of order less than 1, this cannot happen to a 1–base. The second lemma strengthens the idea of subproblems by showing that between lower order steps, bases of any order show this non–decreasing property. While j of the GUB–sets are fixed, the $(j+1)$–base acts like a 1–base. It is the 1–base of this subproblem.

Lemma 5–2: Between consecutive steps of order j or less, $(j+1)$–bases are non–decreasing.

Proof: Let b^{j+1} and $\overline{b}^{j+1}$ be two $(j+1)$–bases occurring between consecutive j–bases, b^{j+1} occuring first. Steps of order $(j+2)$ through m have no effect on $(j+1)$–bases, hence $(j+1)$–bases can only be changed by steps of order $(j+1)$ or less. By construction, there are no steps of order less than $(j+1)$ between b^{j+1} and $\overline{b}^{j+1}$. Any difference between them is the result of $(j+1)$–steps. These only increment some element of the $(j+1)$–base, hence, $b^{j+1} \leq \overline{b}^{j+1}$.■

The third lemma is an important corollary of the second, but is important enough to rate its own discussion. It states that at any point in the algorithm the bases are ordered in a vector sense, that is, $b^j \leq b^{j+1}$. This will not only be useful in proving the theorems, but is the basis for the power of cost-fathoms.

Lemma 5-3: At any stage of the algorithm, $b^j \leq b^{j+1}$ for any j.

Proof. After the last step of order j or less $b^j = b^{j+1}$. By lemma 5-2, b^{j+1} has been non-decreasing since then and b^j has not changed. The result follows.∎

Lemma 5-4 clarifies the relationship between the position currently being considered and the present bases.

Lemma 5-4: At any stage of the algorithm, b^m is the current position.

Proof. By definition, the current position is the position immediately following the last step. If the last step was of order j, the current position is b^j. But after each step, b^m is set to b^j, regardless of j. Hence, b^m is always the current position.∎

$K(u)$ is the cost operator, returning the cost vector associated with the position u. The ability to cost fathom partially rests on the fact that this operator is montonically non-decreasing.

Lemma 5–5: If $u \leq u'$ in a vector sense, then $K(u) \leq K(u')$ in a vector sense.

Proof: It is assumed that the costs have been ordered, non–decreasing, in each GUB–set. Hence, moving further out in any GUB–set cannot decrease the cost associated with that GUB–set. In each component K is non–decreasing. Hence, as a vector, if $u \leq u'$, then $K(u) \leq K(u')$.∎

The next task is to show that the algorithm without fathoms does not cycle, that is, does not consider any position twice. Once this is established, it is clear that it will not cycle with any fathoming scheme. This follows by the subsequence argument mentioned above, namely, if no position is repeated in the sequence, none can be repeated in any subsequence.

Theorem 5–1: The algorithm, without fathoms, does not cycle.

Proof: Let u be some position considered by the algorithm and u' be a position considered by the algorithm after u. Let s denote the lowest order step that takes place between positions u and u'.

If $s = 1$, then, by lemma 5–1, the 1–step incremented $u_{r_1(u)}$. Since 1–bases are non–decreasing, u' will have at least this difference at index $r_1(u)$. Hence, $u \neq u'$.

If $s > 1$, then, by lemma 5–2, the s–bases are non–decreasing between u and u'. Further, there has been at least one s–step, changing the element indexed $r_s(u)$. Hence, $u \neq u'$.∎

In general implicit enumeration it is clear that if no fathoms were allowed, the resultant algorithm would be exhaustive enumeration. Analysis of the partial solution recording method shows this. In this algorithm, the completeness of the exhaustive search is not so clear. As steps are made the order associated with each GUB–set can change. This creates the possibility that positions can be stepped over without being enumerated explicitly or implicitly. In fact, such skipping does not happen.

The proof of this fact is constructive in that it delivers the exact set of base vectors that should be in effect when some position, u, is considered. This is very helpful in debugging computer programs designed to carry out this algorithm. If an errant program is not discovering an optimal solution, the following procedure can be used to pinpoint the step at which the optimal solution was incorrectly enumerated implicitly rather than explicitly.

Theorem 5–2: Allowing no cost or feasibility fathoms, the algorithm considers every position explicitly.

Proof: Let u be some position that should be considered, that is, $1 \leq u_i \leq n_i$ for $i = 1, \ldots, m$.

Select from the sequence of 1–bases generated by the algorithm

$$\bar{b}^1 = \max\{ b^1 : b^1 \leq u \}.$$

Since the initial 1-base is $(1111)^t$ and the 1-bases are non-decreasing with none equal, this maximum is well defined.

Next choose

$$\bar{b}^2 = \max\{ b^2 : b^2 \leq u \text{ and } b^2 \text{ follows } \bar{b}^1 \},$$

where "follows" means in the sequence of the algorithm. Note that the initial 2-base after $\bar{b}^1$ is $\bar{b}^1$. By construction of $\bar{b}^1$, the 1-base following $\bar{b}^1$ exceeds u at the element indexed $r_1(\bar{b}^1)$. If $\bar{b}^2$ followed this 1-base, by lemma 5-3, $\bar{b}^2$ would also exceed u at this component, which would contradict its definition. Hence, $\bar{b}^2$ occurs between $\bar{b}^1$ and the next 1-base. By lemma 5-2, between $\bar{b}^1$ and the next 1-base the 2-bases are non-decreasing. Further, no two are equal. Hence the maximum is well defined here also.

In general, for $j = 2, 3, \ldots, m$, define

$$\bar{b}^j = \max\{ b^j : b^j \leq u \text{ and } b^j \text{ follows } \bar{b}^{j-1} \}.$$

By the same reasoning as for $j = 2$, these maxima are well defined.

It will now be shown by induction that $\bar{b}^m = u$. This will imply that the algorithm considers u since, by construction, it considers $\bar{b}^m$.

<u>*Claim*</u>: $\bar{b}^j_k = u_k$ for $k \in R^j(\bar{b}^j) = R^j(u)$.

(Recall that $R^j(u)$ is the set of indices of the j lowest cost elements corresponding to the position u.)

Anchor Step; $j = 1$:

(Case 1) Assume $\bar{b}^1_{r_1(\bar{b}^1)} \neq n_{r_1(\bar{b}^1)}$. That is, assume the next 1-step will not end the algorithm by running up against the end of the row.

First it will be shown that $\bar{b}^1_{r_1(\bar{b}^1)} = u_{r_1(\bar{b}^1)}$.

By construction, $\bar{b}^1 \leq u$, so by lemma 5-5, $K(\bar{b}^1) \leq K(u)$. By the assumption for case 1, there exists a 1-base after $\bar{b}^1$, call it $\bar{\bar{b}}^1$. By construction of $\bar{b}^1$, it follows that $\bar{\bar{b}}^1_{r_1(\bar{b}^1)} > u_{r_1(\bar{b}^1)}$. Since steps change bases by integral increments,

$$\bar{b}^1_{r_1(\bar{b}^1)} = u_{r_1(\bar{b}^1)}. \tag{1}$$

So clearly,

$$K_{(1)}(\bar{b}^1) = K_{r_1(\bar{b}^1)}(\bar{b}^1) = K_{r_1(\bar{b}^1)}(u). \tag{2}$$

Recalling the degeneracy resolving convention,

$$K_{(1)}(\bar{b}^1) \prec K_{(j)}(\bar{b}^1) \quad , j = 2, 3, \ldots, m \tag{3}$$

Again, since $\bar{b}^1 \leq u$,

$$K_{r_j(\bar{b}^1)}(\bar{b}^1) \leq K_{r_j(\bar{b}^1)}(u) \quad , j = 1, 2, \ldots, m. \tag{4}$$

Assembling (2), (3) and (4) gives

$$K_{r_1(\bar{b}^1)}(u) \prec K_{r_j(\bar{b}^j)}(u); \quad j = 2, 3, \ldots, m.$$

Also,

$$\{ r_j(\bar{b}^1), j = 2, 3, \ldots, m \} = \{ 1, 2, \ldots, m \} - \{ r_1(\bar{b}^1) \}$$

so

$$K_{r_1(\bar{b}^1)}(u) \prec K_j(u) \quad , j \neq r_1(\bar{b}^1).$$

Hence, by definition, $r_1(\bar{b}^1) = r_1(u)$.

(Case 2) Now assume $\bar{b}^1_{r_1(\bar{b}^1)} = n_{r_1(\bar{b}^1)}$. By definition of positions to be considered, $u_{r_1(\bar{b}^1)} \leq n_{r_1(\bar{b}^1)}$ so

$$\bar{b}^1_{r_1(\bar{b}^1)} \leq u_{r_1(\bar{b}^1)} \leq n_{r_1(\bar{b}^1)} = \bar{b}^1_{r_1(\bar{b}^1)}$$

and

$$u_{r_1(\bar{b}^1)} = n_{r_1(\bar{b}^1)} = \bar{b}^1_{r_1(\bar{b}^1)}.$$

Now use (2), (3) and (4) as above to show that $r_1(u) = r_1(\bar{b}^1)$. Hence, $\bar{b}^1_k = u_k$ for $k \in R^1(\bar{b}^1) = R^1(u)$, and the result holds for $j = 1$.

Inductive Step: Assume $\bar{b}^{j-1}_k = u_k$ for $k \in R^{j-1}(\bar{b}^{j-1}) = R^{j-1}(u)$. It is to be shown that $\bar{b}^j_k = u_k$ for $k \in R^j(\bar{b}^j) = R^j(u)$.

(Case 1) Assume $\bar{b}^j_{r_j(\bar{b}^j)} \neq n_{r_j(\bar{b}^j)}$, that is, that the next j-step does not hit the end of a row.

By construction, $\bar{b}^j$ occurred between $\bar{b}^{j-1}$ and the next $(j-1)$-base, in the sequence of the algorithm. By lemma 5-1, $\bar{b}^{j-1}_k = \bar{b}^j_k$ for $k \in R^{j-1}(\bar{b}^{j-1})$. By lemmas 5-3 and 5-5 these elements also have the same ranking so $R^{j-1}(\bar{b}^{j-1})$ is a subset of $R^j(\bar{b}^j)$. Furthermore, for any i and v, the cardinality of $R^i(v)$ is i. Hence, the relationship between $R^{j-1}(\bar{b}^{j-1})$ and $R^j(\bar{b}^j)$ is

$$R^j(\bar{b}^j) = R^{j-1}(\bar{b}^{j-1}) \bigcup \{ r_j(\bar{b}^j) \}.$$

From the inductive hypothesis and the fact immediately above,

$$\bar{b}^j_k = u_k \text{ for } k \in R^{j-1}(\bar{b}^{j-1}) = R^{j-1}(u).$$

By the assumption of case 1, the next j–base after $\bar{b}^j$ will come as the result of a j–step rather than a step of lower order. By construction, this new j–base violates $b^j \leq u$. By an argument analogous to that made for the anchor step, $\bar{b}^j_{r_j(\bar{b}^j)} = u_{r_j(\bar{b}^j)}$.

Hence, $\bar{b}^j_k = u_k$ for $k \in R^{j-1}(\bar{b}^{j-1}) \bigcup \{ r_j(\bar{b}^j) \} = R^j(\bar{b}^j)$. By the same argument used in the anchor step, it can now be shown that $r_j(\bar{b}^j) = r_j(u)$, so $R^j(\bar{b}^j) = R^j(u)$.

(Case 2) Assume $\bar{b}^j_{r_j(\bar{b}^j)} = n_{r_j(\bar{b}^j)}$. Then clearly,

$$\bar{b}^j_{r_j(\bar{b}^j)} \leq u_{r_j(\bar{b}^j)} \leq n_{r_j(\bar{b}^j)} = \bar{b}^j_{r_j(\bar{b}^j)}.$$

Coupling this with the inductive assumption, $\bar{b}^j_k = u_k$ for $k \in R^j(\bar{b}^j)$. There is left but to show that $R^j(\bar{b}^j) = R^j(u)$. For the same reasons as for case 1, $\bar{b}^j_k = u_k$ for $k \in R^{j-1}(\bar{b}^j)$. However, it is also true that $\bar{b}^j_{r_j(\bar{b}^j)} = u_{r_j(\bar{b}^j)}$ from above. Hence, the result follows.

Now, by the inductive argument above, $\bar{b}_k^m = u_k$ for $k \in R^m(\bar{b}^m) = \{1, 2, \ldots, m\}$, that is, $\bar{b}^m = u$. By construction, $\bar{b}^m$ was considered by the algorithm, so u was considered also.■

The different forms of fathoming must now be investigated to determine that they do not cause the algorithm to fail to consider any branch.

(1) Feasibility fathom: Given a partial solution S, a feasibility fathom takes place when it can be determined that no completion of S is feasible (see Chapter 6 for details on how this is done). When this occurs, the algorithm dictates that a step take place of order $|S|$.

Corollary 1: Allowing feasibility fathoms, the algorithm will not skip all optimal solutions.

Proof: Let $j = |S|$, for notational ease. Consider the sequence of positions investigated by the algorithm without feasibility fathoms. Carrying out a j–step at a time when no completion of S is feasible causes some elements of this sequence to be skipped. These positions all fall between the j–base in effect when the feasibility fathom took place and the next j–base in the sequence of the algorithm. During this time, only steps of order higher than j would have taken place. By lemma 5–1, all these positions would have had S as a subset of their partial solutions. The hypothesis is that no completion of S is feasible, so none of these positions is feasible. Hence, they cannot be optimal.∎

(2) Cost fathom: As described in Chapter 3, a cost fathom occurs when the cost associated with a j–base, $\overline{K}(b^j)$, is not less than the present best cost, $\overline{z}$. Note that this means the cost associated with all of b^j, not just those elements in GUB–sets in S. The algorithm dictates that when this condition exists, a step of order $|S| - 1$ is taken.

Corollary 2: Allowing cost fathoms, the algorithm will not skip all optimal solutions.

Proof: Let $j = |S|$. Claim: $\overline{K}(b^j)$ is a lower bound for the costs of all positions encountered before the next $(j - 1)$–step.

Between the time of the cost fathom and what would have been the next j–step, the current position,b^m by lemma 5–4, is not less than b^j, by lemma 5–3. Hence, by lemma 5–5, they all have higher cost than b^j. Since $\overline{K}(b^j) \geq \overline{z}$, so is the cost of any of these positions. Hence, none of them can be optimal.

Further, before the next $(j - 1)$–step, the j–bases are non–decreasing by lemma 5–2 and, by lemma 5–5, so are their costs. By the argument above, all positions considered between each of these j–bases has higher cost than b^j. Hence, until the next $(j - 1)$–step, all positions have cost not less than $\overline{K}(b^j) \geq \overline{z}$. They cannot be optimal since a solution of not greater cost is already known.∎

When a j–step leads to a feasible completion that becomes the new incumbent, the algorithm allows a $(j-1)$–step. This follows directly from corollary 2. As soon as the new incumbent is recorded, $\overline{K}(b^j) \geq \overline{z}$. Hence, by corollary 2 a cost fathom occurs.

At some point in the algorithm a cost or row fathom will occur on a 1–step. This is the signal that the algorithm has completed its search and that the current incumbent is optimal. It must be shown that when this happens, no position that has not been considered need be considered.

Corollary 3: On a cost or row fathom on a 1–step, the current incumbent is an optimal solution.

Proof: Consider a cost fathom on a 1–step. Recall that 1–bases are non–decreasing. Hence, if the current 1–base has cost not less than the incumbent, so will all subsequent 1–bases. By lemmas 5–3, 5–4 and 5–5, then, so will all positions considered subsequently.

When a 1–step attempts to move a position beyond the end of a row, the algorithm has considered (implicitly) all positions. In either this case or that above, the search may be terminated.∎

• Tie–Breaking

As mentioned in Chapter 3, the method used to break ties in cost values must be given some care. For this discussion assume the variables in each GUB–set have been ordered so that for $i = 1, \ldots, m$ and $j = 1, \ldots, n_i - 1, c_{ij} \leq c_{i(j+1)}$. Let the order relation, with some tie–breaking procedure, be denoted by $\prec$.

For an order relation to result in an algorithm that does not consider positions more than once it must have the following property. The order relation is said to be **consistent** if and only if

$$c_{ij} \prec c_{kl} \text{ implies } c_{ij} \prec c_{k(l+1)}, \quad \forall i, j, k, l.$$

The least index rule defines the following order relation.

$$c_{ij} < c_{kl} \text{ implies } c_{ij} \prec c_{kl}$$

$$c_{ij} = c_{kl} \text{ and } i < k \text{ implies } c_{ij} \prec c_{kl}$$

$$\text{otherwise, } c_{kl} \prec c_{ij}.$$

Lemma 5–6: The least index rule order relation is consistent.

Proof: Assume $c_{ij} \prec c_{kl}$. If $c_{ij} < c_{kl}$, since $c_{kl} \leq c_{k(l+1)}$ by assumption, then $c_{ij} < c_{k(l+1)}$. Hence, $c_{ij} \prec c_{k(l+1)}$.

If $c_{ij} = c_{kl}$ and $c_{kl} < c_{k(l+1)}$ then $c_{ij} < c_{k(l+1)}$ and $c_{ij} \prec c_{k(l+1)}$.

If $c_{ij} = c_{kl} = c_{k(l+1)}$ then since $c_{ij} \prec c_{kl}$ it must be true that $i < k$. Hence, $c_{ij} \prec c_{k(l+1)}$.∎

CHAPTER 6

CONSTRAINT HANDLING

The statement of the algorithm in Chapter 3 includes references to problem solution "by inspection." This chapter will detail how this can be accomplished algorithmically. Most of the efficiency of implicit enumeration comes from cleverness employed in these inspections. The more subproblems that can be solved by inspection the fewer the separations that are needed and the quicker the process. However, the deeper the inspection, the more time consuming the inspections become. The trade–off is between a greater number of quicker iterations or fewer slower iterations. The techniques presented here will substantially follow the lines of those for general additive algorithms. However, in most cases, the special structure of the MCIP will allow revisions to improve efficiency, substantially in some cases.

The question to be answered is the following. Given a partial solution, S, can an optimal solution be a completion of S? (Recall that a completion is an assignment of values to all variables not fixed by S.) If it cannot, there is no need to consider this partial solution further; S is thereby fathomed so backtracking leads to a different partial solution to be analyzed. If an optimal solution could be a completion of S another question must be asked.

Can the best possible completion of S be discovered easily? If so, and if it is better than the best solution identified so far, this new incumbent solution must be stored until such time as it can be determined whether or not this solution is actually optimal. In this case, as above, the present partial solution has been fathomed (examined as much as is necessary) and another should be investigated. If the best completion of S cannot be found easily, S is augmented with more information. Following is a description of some possible methods for answering these questions.

- **Binary Infeasibility**

The binary infeasibility test is a method of determining if a partial solution, S, has no feasible completion. If S indeed has no feasible completion, it clearly cannot have an optimal solution as a completion. This test involves one constraint at a time, its greatest limitation.

Recall that constraint k can be stated as

$$b_k + \sum_{i=1}^{m} \sum_{j=1}^{n_i} a_{kij} x_{ij} \geq 0.$$

In the general formulation, without GUB–constraints, consider $n_i \equiv 1$ and $m = n$, the number of variables.

Define $b_k^S = b_k + \sum_{i \in S} a_{kiu_i}$, where u is the partial position associated with S. The corresponding constraint for the subproblem defined by S is then

$$b_k^S + \sum_{i \notin S} \sum_{j=1}^{n_i} a_{kij} x_{ij} \geq 0. \tag{1}$$

In general, the maximum value that the left hand side of this inequality can attain is reached when $x_{ij} = 0$ for all $\{i, j : a_{kij} < 0\}$ and $x_{ij} = 1$ for all $\{i, j : a_{kij} \geq 0\}$. This maximum value can be expressed as

$$b_k^S + \sum_{i \notin S} \sum_{j=1}^{n_i} a_{kij} x_{ij} \leq b_k^S + \sum_{i \notin S} \sum_{j=1}^{n_i} \max(0, a_{kij}). \tag{2}$$

Hence, if the right hand side in (2) is less than zero, no completion of S can be feasible for constraint k. Therefore, the binary infeasibility test checks this expression one constraint at a time, and if the right hand side in (2) is negative for any k, S has no feasible completion and the algorithm should backtrack.

This test should never be made for the implicit GUB–constraints. All solutions considered satisfy these constraints. Hence, any partial solution has a completion feasible for these constraints. This reduces the computation necessary to carry out the test.

Beyond this, the known GUB structure can be used to strengthen these tests for use in this algorithm. Any feasible completion of S will have exactly one variable in each free GUB–set fixed at 1. This fact leads to a tighter upper bound on the left side of (1) for GUB problems. Since exactly one variable has the value one in any GUB–set i and in any feasible solution for a constraint k,

$$\sum_{j=1}^{n_i} a_{kij}x_{ij} \leq \max_j(a_{kij}), \forall i, \forall k.$$

Hence,

$$b_k^S + \sum_{i \notin S} \sum_{j=1}^{n_i} a_{kij}x_{ij} \leq b_k^S + \sum_{i \notin S} \max_j(a_{kij}). \tag{3}$$

It should be clear that the bound in (3) is tighter than the bound in (2).

The added strength comes from two sources. First, if all the $a_{kij} < 0$ for some GUB–set and some constraint, $\sum_{j=1}^{n_i} \max(a_{kij}, 0) = 0$, but $\max_j(a_{kij}) < 0$. This exploits the fact that one element of this GUB–set must be fixed at 1. Second, when more than one element of some GUB–set has a positive coefficient in some constraint, the tighter expression includes only the largest of these coefficients. The more general expression, (2), includes all positive coefficients. The tighter expression reflects the fact that only one variable in each GUB–set is non–zero.

Ideally, all partial solutions with no feasible completions would fail the binary infeasibility test. As will be seen in the section on surrogate constraints, this is not the case. When a partial solution without any feasible completions does not fail, time will be wasted investigating this partial solution further. Thus, these tighter bounds are advantageous because they force a greater number of partial solutions having no feasible completions to fail the test.

- **Conditional Binary Infeasibility**

This technique is closely related to binary infeasibility. For a particular constraint, k, a partial solution, S, may have feasible completions, but it is possible that some particular variable is assigned the value 1 in all of these completions. Alternatively, a certain variable may have the value 0 in all completions. In general additive algorithms, when the conditional binary infeasibility test determines that this is the case, the variable involved can be fixed at this value and added to the current partial solution.

For an example from general binary programming consider the problem

$$\begin{aligned} &\text{maximize} && w + x + y + z \\ &\text{subject to:} && -w - 2x + y + z \geq 0 \qquad (5) \\ &&& w, x, y, z \in \{0, 1\}. \end{aligned}$$

Given a partial solution $\{w = 1\}$, for (5) to hold, x must be 0.

In general binary programming, conditional infeasibility can be tested for in a given constraint, k, and variable $x_{i_0 j_0}$, by considering the sign of

$$b_k^S + \sum_{(i,j) \notin S} \max(0, a_{kij}) - |a_{k i_0 j_0}|. \qquad (6)$$

If (6) is less than zero then $a_{k i_0 j_0} < 0$ implies that $x_{i_0 j_0} = 0$ for any feasible binary completion of S and $a_{k i_0 j_0} > 0$ implies that $x_{i_0 j_0} = 1$ in any feasible completion.

As with the binary infeasibility test, this can be made more powerful by using the structure of the GUB problem. If $a_{ki_0j_0} < 0$ and

$$b_k^S + \sum_{\substack{i \notin B \\ i \neq i_0}} \max_j(a_{kij}) + a_{ki_0j_0} < 0$$

then $x_{i_0j_0} = 0$ in all feasible binary completions of S. If $a_{ki_0j_0} > 0$ and

$$b_k^S + \sum_{\substack{i \notin B \\ i \neq i_0}} \max_j(a_{kij}) + \max_{j \neq j_0}(a_{ki_0j}) < 0$$

then $x_{i_0j_0} = 1$ in all feasible binary completions of S. Both of these expressions are not greater than (6) and hence this is a stronger test.

Though this test is more powerful than the test involving (6), this technique is not nearly so useful in this algorithm as it is in general binary programming. In the latter, if the test in (6) indicates some variables are conditionally fixed for a given partial solution, they can be added to the partial solution at their forced values and "underlined" (see [Geoffrion, 1967–1]). This augmentation is not so easily done in the framework of this algorithm. The augmentation rules are much more structured. Variables are augmented by GUB–sets and not individually. If the conditional tests indicate that some variable should be set to 1, that does fix the entire GUB–set, but if it is to be forced to 0 it is still not known where the 1 falls in that GUB–set. To further complicate matters, the order in which the GUB–sets are augmented is quite structured so that a conditional fixing of an arbitrary GUB–set is not allowable.

On the other hand, when the conditional tests indicate that a variable should be fixed for the life of this partial solution, that information can be stored. Then, when executing a step, the new configuration of 1's can be checked for violation of any of these conditional fixings. If any conflicts are found, the new partial solution can be fathomed immediately, just as if it had failed the test in (3). This technique was implemented and did reduce the number of iterations required to solve problems. Unfortunately, the computational time required to do the conditional tests and checking during the stepping procedure was great enough to offset this advantage.

- **Surrogate Constraints**

When the above techniques determine that a partial solution has no binary feasible completion the evidence in conclusive. However, when they do not determine this, it is not necessarily true that there exists a feasible completion. Consider the following example:

$$\begin{aligned} \text{minimize} \quad & x + y \\ \text{subject to:} \quad & -1 + x + y \geq 0 \\ & x - y \geq 0 \\ & x, y \in \{0, 1\}. \end{aligned}$$

By inspection, this problem has two feasible solutions, (1,0) and (1,1), of which (1,0) is optimal. Consider a situation in an additive type solution of this problem where the partial solution is $\{x = 0\}$. Even though there are no feasible completions to this partial solution, the binary infeasibility test does not indicate this. The first constraint finds that it can be satisfied by some completion, (0,1), and the second constraint finds that it can also, (0,0). The lack of intersection in the sets of feasible completions for the two constraints is obvious in this small example, but in larger problems this same sort of exclusivity occurs and is much more difficult to uncover.

This weakness in the tests exists because the tests consider only one constraint at a time. They determine if each constraint has a non–empty set of feasible completions to the partial solution, without regard to whether or not any of these feasible sets intersect. The concept of surrogate constraints arose shortly after Balas' introduction of additive type algorithms [Balas,1965-1], first by Glover [Glover, 1967-1] and then strengthened by Geoffrion [Geoffrion, 1969-1]. This idea addresses the question of intersection of the feasible sets. The discussion will center on Geoffrion's version.

In the framework of an additive algorithm assume the constraints are denoted by $b + Ax \geq 0$. A surrogate constraint is a non–negative linear combination of the existing constraints, that is,

$$\mu^{t}(b + Ax) \geq 0 \qquad (SC)$$

where $\mu \in \Re^{\kappa}$ and $\mu \geq 0$. For any x such that $b + Ax \geq 0, \mu^{t}(b + Ax) \geq 0$ by the non–negativity of μ. Hence, (SC) does not eliminate anything from the feasible region. However, by combining information from many constraints, (SC) may allow a better binary infeasibility test later in the algorithm.

Let S be the partial solution in effect at some point in an additive algorithm run. Given this information it is determined that a μ should be chosen to construct a surrogate constraint. In some sense, a "best" μ should be found. Geoffrion suggests an objective that proves to be very powerful:

$$\min_{\mu} \max_{x} \mu^t(b + Ax) + \bar{z} - cx \tag{7}$$

where $\bar{z}$ is the incumbent objective value.

Optimizing this objective accomplishes two goals: it determines if S has no feasible completions and also if there are no feasible completions with objective better than the incumbent. Fixing those elements of x that are determined by S and rearranging terms, (7) becomes

$$\min_{\mu} \max_{x_{ij}:i \not\in S} \sum_{k=1}^{K} \mu_k(b_k^S + \sum_{i \not\in S} \sum_{j=1}^{n_i} a_{kij}x_{ij}) + \bar{z} - z^S - \sum_{i \not\in S} \sum_{j=1}^{n_i} c_{ij}x_{ij}. \tag{8}$$

Lemma 6—1: If the partial solution S has no feasible completion, (8) will have an optimal value less than zero.

Proof: If S has no feasible completion, then for some k,

$$b_k^S + \sum_{i \not\in S} \sum_{j=1}^{n_i} a_{kij}x_{ij} < 0$$

for all possible x. The set of possible x is finite so

$$\beta = \max_{x}(b_k^S + \sum_{i \notin S} \sum_{j=1}^{n_i} a_{kij}x_{ij}) < 0.$$

Let

$$\gamma = |\bar{z} - z^S - \sum_{i \notin S} \sum_{j=1}^{n_i} c_{ij}x_{ij}|.$$

Let $\mu_k = -\gamma/\beta + 1$. Set all other elements of μ to 0. The optimand in (8) is less than zero for this μ, hence, the minimum over all $\mu \geq 0$ is certainly less than zero.∎

Lemma 6—2: If the partial solution, S, has no feasible completion with objective value less than $\bar{z}$, the optimal value of (8) will be not greater than zero.

Proof: If there are no feasible completions to S, this lemma is true by lemma 1. Assume there are feasible completions to S. Let X_f^S be the set of feasible completions of S. By assumption, $\bar{z} - cx \leq 0$ for $x \in X_f^S$. Hence, $\max_{x:x \in X_f^S}(\bar{z} - cx) \leq 0$. Now choose $\mu \equiv 0$ so the optimand of (8) is not greater than zero. Hence, the minimum over all $\mu \geq 0$ is not greater than zero.∎

Lemma 6—3: If S has a feasible completion with objective value better than

$\bar{z}$, the optimal value of (8) will be strictly positive.

Proof: Let $\bar{x}$ be this feasible completion with better objective value. Then $\bar{z} - c\bar{x} > 0$ and $b + A\bar{x} \geq 0$. For any $\mu \geq 0$, $\mu^t(b + A\bar{x}) \geq 0$ so

$$\min_{\mu \geq 0} \sum_{k=1}^{\kappa} \mu_k (b_k^S + \sum_{i \notin S} \sum_{j=1}^{n_i} a_{kij} \bar{x}_{ij}) + \bar{z} - z^S - \sum_{i \notin S} \sum_{j=1}^{n_i} c_{ij} \bar{x}_{ij} > 0.$$

Hence, (8) is greater than zero.∎

Since the hypotheses of lemma 6—1,6—2 and 6—3 are exhaustive, if the optimal value of (8) is strictly positive, the only possibility is that S has a feasible completion with better objective value than $\bar{z}$. If the optimal value of (8) is non-positive, the contrapositive of lemma 6—3 states that S does not have a feasible completion with objective value better than $\bar{z}$.

Consequently, if the value of (8), for some S, is not greater than 0, there is no use investigating S further. If the value is positive, the argmax, x, is a new incumbent. Unfortunately, (8) is a small version of the original binary program, so it is not considerably easier to solve than the original problem. However, a relaxation of (8) is much easier to solve and delivers almost as much information. Following is a derivation of the surrogate constraint LP for the partial solution S for general binary programming adapted from [Geoffrion, 1969-1].

For a given μ, (8) becomes

$$\sum_{k=1}^{K} \mu_k b_k^S + \bar{z} - z^S + \max_{\substack{x_{ij} \in \{0,1\} \\ i \notin S}} \{ \sum_{i \notin S} \sum_{j=1}^{n_i} (\sum_{k=1}^{K} \mu_k a_{kij} - c_{ij}) x_{ij} \}.$$

Now the portion of this that is to be maximized is equal to

$$\max_{\substack{0 \le x_{ij} \le 1 \\ i \notin S}} \{ \sum_{i \notin S} \sum_{j=1}^{n_i} (\sum_{k=1}^{K} \mu_k a_{kij} - c_{ij}) x_{ij} \} \tag{9}$$

because the obvious optimal solution to this maximization problem is to choose $x_{ij} = 0$ if $\sum_{k=1}^{K} \mu_k a_{kij} - c_{ij} \le 0$ and $x_{ij} = 1$ otherwise. The optimal values will not take on fractional values. Taking the dual of (9) gives

$$\min_{\substack{w_{ij} \ge 0 \\ i \notin S}} \{ \sum_{i \notin S} \sum_{j=1}^{n_i} w_{ij} : w_{ij} \ge \sum_{k=1}^{K} \mu_k a_{kij} - c_{ij} \}.$$

Combining the parts gives the LP

$$\begin{array}{ll} \text{minimize} & \sum_{k=1}^{K} \mu_k b_k^S + \bar{z} - z^S + \sum_{i \notin S} \sum_{j=1}^{n_i} w_{ij} \\ \text{subject to:} & w_{ij} \ge \sum_{k=1}^{K} \mu_k a_{kij} - c_{ij} \quad , i \notin S \\ & \mu, w \ge 0. \end{array} \tag{LP_S}$$

Let the optimal value of (LP_S) be $v(LP_S)$. By lemmas 6—1, 6—2 and 6—3, if $v(LP_S) \le 0$, S is fathomed. If $v(LP_S) > 0$, and the dual variables (now x_{ij}'s) are integral, they denote the optimal completion of S and should be recorded as a new incumbent. However, due to the relaxation of $x_{ij} \in \{0,1\}$ to $0 \le x_{ij} \le 1$ the dual variables may be fractional. It is known that

S should be explored further, but the dual variables are not the optimal completion of S. In this case, the optimal primal μ_k's should be used to construct a surrogate constraint and S should be augmented. This surrogate constraint will help make future binary infeasibility tests more powerful.

To this point, the entire discussion of surrogate constraints has been relevant to general binary programs. Next it will be specialized to the MCIP. The form of the surrogate constraint LP that has been discussed is applicable to the GUB case with the exception that the implicit GUB constraints are left out of the analysis. This ommission is easily corrected.

Consider the GUB–constraint

$$\sum_{j=1}^{n_i} x_{ij} = 1.$$

It can be split into two inequality constraints,

$$1 - \sum_{j=1}^{n_i} x_{ij} \geq 0,$$

$$-1 + \sum_{j=1}^{n_i} x_{ij} \geq 0,$$

that fit the general constraint formulation used in this problem. Though not efficient, it is possible to include $2m$ constraints of this type in the general constraints, $b + Ax \geq 0$, and solve the as a general binary program. Were this done, LP_S would find multipliers for these constraints. Let η_i be the multiplier for the constraint $1 - \sum_{j=1}^{n_i} x_{ij} \geq 0$ and θ_i be the multiplier for the constraint $-1 + \sum_{j=1}^{n_i} x_{ij} \geq 0$. Recall that in the objective function of LP_S the coefficients are b^S. For η_i, if GUB–set i is fixed by S, then $b^S = 0$. If $i \not\in S$ then $b^S = 1$. For θ_i, if $i \in S$ then $b^S = 0$ and if $i \not\in S$, $b^S = -1$. In the constraints, η_i appears only in those corresponding to variables in GUB–set i. In those GUB–sets it has coefficient -1. The same is true for θ_i except that the coefficients are 1. Hence, the LP used to calculate surrogate constraints for the GUB case is stated:

$$\begin{aligned} &\text{minimize} && \sum_{k=1}^{\kappa} \mu_k b_k^S + \bar{z} - z^S + \sum_{i \not\in S} \sum_{j=1}^{n_i} w_{ij} + \sum_{i \not\in S} (\eta_i - \theta_i) \\ &\text{subject to:} && w_{ij} \geq \sum_{k=1}^{\kappa} \mu_k a_{kij} - c_{ij} + \theta_i - \eta_i, i \not\in S, j = 1, 2, \ldots, n_i \\ & && \hspace{20em} (GLP_S\text{-}1) \\ & && w, \mu, \theta, \eta \geq 0. \end{aligned}$$

The primary difficulty with these LP's is that they are quite large. For a MCIP with 400 variables, 20 general constraints, 20 GUB–sets, the GLP_S–1 formulation for the partial solution $S = \emptyset$ is 400 by 460 without slacks. In general, if the values above are n, κ, m, the largest GLP_S–1 is n by $(n + \kappa + 2m)$. Following is a derivation of a surrogate constraint LP special to this GUB problem that has maximum size n by $\kappa + 2m$. For the example above, this LP would be 400 by 60. The improvement is based on the observation that, for the GUB structured problem, the variables w_{ij} are redundant and can be eliminated. They are the dual variables of the constrants $x_{ij} \leq 1$. These constraints are redundant in a problem with constraints $1 - \sum_{j=1}^{n_i} x_{ij} \geq 0$.

To derive the new LP, assume the constraints $-1 + \sum_{j=1}^{n_i} x_{ij} \geq 0$ are included in the general constraints, $b + Ax \geq 0$. Then (9) can be stated

$$\max_{\substack{0 \leq x_{ij} \\ i \notin S}} \{ \sum_{i \notin S} \sum_{j=1}^{n_i} (\sum_{k=1}^{\kappa} \mu_k a_{kij} - c_{ij}) x_{ij} : \sum_{j=1}^{n_i} x_{ij} \leq 1 \}.$$

Taking the dual of this gives

$$\min_{\substack{\eta_i \geq 0 \\ i \notin S}} \{ \sum_{i \notin S} \eta_i : \eta_i \geq \sum_{k=1}^{\kappa} \mu_k a_{kij} - c_{ij} \}.$$

Reconstructing the LP and separating the θ_i from the other μ_k's leaves

$$\text{minimize} \qquad \sum_{k=1}^{\kappa} \mu_k b_k^S + \overline{z} - z^S + \sum_{i \notin S} (\eta_i - \theta_i)$$

$$\text{subject to:} \qquad 0 \geq \sum_{k=1}^{\kappa} \mu_k a_{kij} - c_{ij} - (\eta_i - \theta_i) \quad , i \notin S, j = 1, 2, \ldots, n_i$$

$$(GLP_S\text{-}2)$$

$$\mu, \eta, \theta \geq 0.$$

As expected, this is precisely GLP_S-1 with the w_{ij} eliminated. Non-trivial variables now number only $(\kappa + 2m)$, 60 in the example.

Once slacks are added to the problem, the basis of this formulation is n by n, 400 by 400 in the example. By operating on the dual of this problem, the basis can be reduced to $(\kappa + 2m)$ by $(\kappa + 2m)$, 60 by 60 in the example.

Using the concept of continuous variable GUB programming ([Dantzig & Van Slyke, 1964–1]) this could be reduced to a working basis of κ by κ , 20 by 20 in the example. The full basis formulation can be stated

$$\text{maximize} \quad \sum_{i \notin S} \sum_{j=1}^{n_i} -c_{ij} x_{ij} + z - z^S$$

$$\text{subject to:} \quad b_k^S + \sum_{i \notin S} \sum_{j=1}^{n_i} a_{kij} x_{ij} \geq 0 \quad , k = 1, 2, \ldots, \kappa \qquad (GLP_S\text{-}3)$$

$$\sum_{j=1}^{n_i} x_{ij} = 1 \quad , i \notin S$$

$$x_{ij} \geq 0.$$

This is virtually the original GUB–constrained binary program with integrality relaxed. However, by understanding the relationship between GLP_S-3 and the strongest surrogate constraint, a great deal more information can be read from its solution.

Using any form of the surrogate constraint LP, it is still necessary to set up and solve many LP'^{s} during the solution of a MCIP. On the surface this seems an overly time consuming proposition. The technique becomes efficient when the relationships between the various LP'^{s} solved for a particular adaptive problem are used to reduce set–up and solution times. The relationships exploited are different for formulations GLP_S–2 and GLP_S–3. The formulation GLP_S–2 has a more exploitable structure so that it warrants further consideration despite its larger basis.

Using GLP_S–2 for this GUB structured implicit enumeration algorithm, only one LP need be analyzed in detail, $GLP_\emptyset$–2. For any S, the constraints in GLP_S–2 are a subset of the constraints in $GLP_\emptyset$–2. To see this, recall that there is one constraint for each free variable. In $GLP_\emptyset$–2 all variables are free so there is a constraint for each variable. From the formulation it can be seen that the content of each constraint does not depend on S. S only determines which constraints exist. The objective row, however, does depend on S. The coefficients are b^S.

Geoffrion suggests a way of moving from one surrogate constraint LP to the next. The LP is only set up once, in the $S = \emptyset$ form; then for each iteration certain adjustments are made. For any row that should not belong to the LP relative to the given S, one of the following takes place. If x_{ij} should be fixed at zero because of S, the slack for the constraint corresponding to x_{ij} is tagged as free (as opposed to the usual non–negativity). If x_{ij} should be fixed at one because of S, w_{ij} is tagged as free. The objective row does not need to be altered. Recall that the variable w_{ij} is the dual variable for the constraint $x_{ij} \leq 1$. Freeing w_{ij} forces it into the basis (see [Geoffrion, 1969–1]) assuring, by complementary slackness, that $x_{ij} = 1$. The slack for the constraint corresponding to x_{ij} is the dual variable for the constraint $x_{ij} \geq 0$ so freeing it forces $x_{ij} = 0$ in any solution. In this way the program automatically updates b^S.

The optimal basic feasible solution from the last LP is altered to be basic feasible for the new S. At this point the LP is ready to solve the new problem. This advanced basis makes solving subsequent LP's very quick.

This procedure will not work for GLP_S–2. The w_{ij} have been eliminated so that they cannot be tagged free or non-negative. However, an alternative method can be used that has been very effective. It is necessary to recompute the objective row coefficients to reflect b^S for the current S. This essentially eliminates the rows that should not be in the formulation for this S as follows. In GLP_S–2, the substitution, $\alpha_i = \eta_i - \theta_i$ is made, with α_i a free variable. Then α_i would have a coefficient of 1 in the objective row and -1 in any row corresponding to GUB-set i. When GUB-set i is fixed by S, the objective row coefficient of α_i is zero. Hence, every row that should not be in the formulation contains a free variable that has a zero coefficient in the objective row. Any solution of the other constraints can be made feasible in these rows with appropriate values of these α_i without affecting the objective value.

The efficiency of this technique lies in the fact that, since only the objective row is altered from one LP to the next, the optimal basic feasible solution to the last LP is basic feasible to the next. The LP routine starts with a very advanced basis so that, in practice, it has required only one or two pivots to go optimal for subsequent LP's.

Similar techniques exist for GLP_S–3. Here the right hand side is altered rather than the objective row. The optimal basis from the last LP is now dual feasible but probably not primal feasible. Dual pivots can now be executed to make it feasible. Unfortunately, this requires that the LP routine be able to execute both primal and dual pivots. In most efficient LP codes the data is stored compactly. If it is stored in a column major manner, primal pivots are done very efficiently but dual pivots are not. They require unpacking of all columns rather than just the column of the incoming variable. If data packing is done in a row major manner, dual pivots are efficient and primal pivots are not. If both types of pivots are required, the LP code cannot take advantage of this compact data storage.

For this reason, the code for this algorithm employs the formulation GLP_S–2. If the algorithm were reprogrammed with an LP package that handled both primal and dual pivots efficiently, it would be advisable to use formulation GLP_S–3 due to the smaller sizes of LP's that must be solved for a given MCIP.

CHAPTER 7

DEVELOPMENT AND ANALYSIS OF TWO NEW VERSIONS OF THE ALGORITHM

This chapter presents two new versions of the algorithm, one heuristic and one exact, with discussion of the theoretical and empirical performance of the exact version.

• A Heuristic Version of the Algorithm

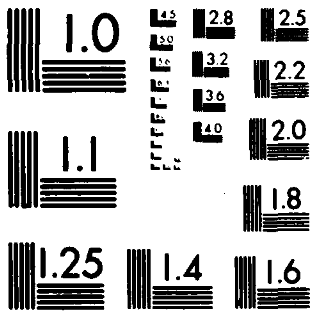
Following is a discussion of a heuristic version of the algorithm that has guaranteed accuracy and, in testing, has never failed to find an optimal solution. This heuristic is then evolved to another exact version of the algorithm that is much faste, in most cases, than the version presented in Chapters 3 and 6.

When, for a given partial solution S, the surrogate constraint LP returns an optimal objective value that is strictly positive, that value is an upper bound on the improvement possible with any completion of S. For example, suppose that the current incumbent has objective $\bar{z} = 846$ and that, for the current partial solution S, the LP's optimal value is 52. Then the best possible new incumbent value that could be attained from a completion of S

AD-A092 691   STANFORD UNIV CA DEPT OF OPERATIONS RESEARCH   F/G 12/1
AN ADDITIVE ALGORITHM FOR THE MULTIPLE CHOICE INTEGER PROGRAM.(U)
OCT 80 J C BEAN   N00014-76-C-0418
UNCLASSIFIED   TR-96   NL

2 of 2

END
DATE
FILMED
1 -8
DTIC


1.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
10.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

would have value $846 - 52 = 794$.

To see that this upper bound is valid, recall from Chapter 6 that the surrogate constraint LP used is the dual of the continuous variable relaxation of the surrogate program defined by S. The relaxation of the integral requirements comes in the primal statement of the program, a maximization problem. Hence, the relaxation increases the optimal value if anything. When the dual is taken, the optimal objective value is not changed so it is still an overestimate of the amount of savings possible with any completion of S.

Consider the following scenario. During a run of the algorithm the current partial solution is S and the surrogate constraint LP returns an optimal objective value of $z_{LP} > 0$. If the dual variables are not all integral, the algorithm dictates that a surrogate constraint be added and the partial solution augmented. The heuristic deals partially with the question of whether of not it is desirable to augment here or not. Consider the amount that could be gained by investigating the current partial solution further. If $\bar{z} = 846$ and $z_{LP} = 3$, is it worth searching for the optimal completion of S that may save up to 3 units out of 846? Three points arise here. First, in many real applications a difference so small really has no significant effect. In fact, there is probably more error in the input data than this. Second, it is quite possible that this optimal completion is not an optimal solution to the original problem, in which case no accuracy is lost by skipping over this completion.

Third, once an augmentation takes place it often takes a great deal of time to get back down to the order from which the augmentation took place. As a result, there is a great deal to be gained, and perhaps nothing to be lost, by avoiding some augmentations.

The heuristic proceeds as follows. Input a factor $p \in [0,1]$. When the algorithm decides it should augment, actually do so only if $z_{LP} > p\bar{z}$. In other words, do not bother to augment if the maximum possible improvement, z_{LP}, is too small ($\leq p\bar{z}$). This may introduce error (deviation from optimality) into the solution, since it is possible that all optimal solutions could be skipped over. However, the power of this technique comes from the fact that the possible errors do not accumulate. For the final $\bar{z}$, the maximum deviation from the optimal objective value is $p\bar{z}$.

To see why this is the maximum error, consider the moment the first optimal solution, with objective value z^*, is skipped over. This occurs when a partial solution, of which this optimal solution is a completion, is incorrectly fathomed because $z_{LP} \leq p\bar{z}$. Because z_{LP} is an upper bound on improvement of the incumbent value for completions of S such as z^*,

$$\bar{z} - z_{LP} \leq z^*.$$

Since $z_{LP} \leq p\bar{z}$

$$\bar{z} \leq z^* + z_{LP} \leq z^* + p\bar{z}.$$

Since this is a minimization problem, $z^* \leq \bar{z}$, so

$$0 \leq \bar{z} - z^* \leq p\bar{z}. \tag{1}$$

The incumbent values after this "skipping over" are strictly decreasing and bounded below by z^*. Hence, if anything, the error will decrease for the remainder of the algorithm.

It is desirable to construct a bound on the possible error in this procedure as a percentage of the correct optimal value. Such a bound is quite easily found. From (1)

$$(1-p)\bar{z} - z^* \leq 0$$

which implies that

$$\frac{\bar{z}}{z^*} \leq \frac{1}{(1-p)}.$$

Let $1 + \rho = 1/(1-p)$, so $p = \rho/(1+\rho)$. The error in the solution is then bounded above by 100ρ percent of the true optimal value, z^*. To ensure 5% accuracy ($\rho = .05$) let $p = .05/(1 + .05) \approx .048$. Numerous problems have been solved using this heuristic, with values for p as high as .5, and all have obtained a true optimal value.

- **A New Exact Version of the Algorithm**

This heuristic procedure can easily be altered to form an algorithm that ensures optimality, but that performs much better than the original version of the algorithm ($p \equiv 0$). In particular, whenever a new incumbent is recorded, save the bases and order in effect at that time. Set p relatively high, say .1 or .2. When the algorithm stops with a final heuristic solution, return to the stored base and order, set $p = 0$ and finish optimally.

Lemma 7–1: If any optimal solution has been skipped over during the heuristic pass, then this (and all other) optimal solution was skipped over *after* the last incumbent was recorded.

Proof: Assume to the contrary that an incumbent, with objective $\bar{\bar{z}}$, is recorded after some optimal solution, with objective value z^*, is skipped over. Denote the objective value for the preceding incumbent by $\bar{z}$. Since z^* was skipped over,

$$\bar{z} - z^* \leq p\bar{z}.$$

Since $\bar{\bar{z}}$ was not skipped over,

$$\bar{z} - \bar{\bar{z}} > p\bar{z}.$$

(Recall that if a new incumbent if recorded, then it is a completion consisting of the dual variables which must be integral. In this case the upper bound on improvement, z_{LP}, is attained.) Combining these gives

$$\bar{\bar{z}} < \bar{z} - p\bar{z} \leq z^*.$$

This contradicts the fact that z^* is the optimal objective value.∎

By lemma 7–1, if the optimal solutions were skipped over, it was done at the end of the heuristic run after obtaining the final incumbent. Hence, going over the end again with $p = 0$ will find an optimal solution.

One technical point must be brought out here. It is assumed that the test for $z_{LP} \leq p\bar{z}$ is made before checking integrality of the dual variables. Otherwise, it is possible that $z_{LP} < p\bar{z}$ and the dual variables are integral. If this is recorded as a new incumbent, lemma 7–1 does not hold and the optimal solution will not necessarily be found in the verification pass.

This version of the algorithm is almost as efficient as the heuristic from which it was derived. In every case tested, the heuristic pass found an optimal solution. When this occurs, the verification pass stops very quickly because of the power of the surrogate constraint LP. If the optimal solution was found on a j–step, the verification part of the algorithm takes j iterations to verify its optimality.

Table 7–1 compares run times for the version of the algorithm in Chapters 3 and 6 and the version with this improvement. The problems used were 10 and 14 team league scheduling problems and 11 and 20 task assembly line balancing problems. The details of these problems will be described later in this chapter. On the scheduling problems the new version did much better. On the assembly line balancing problems they did similarly. In general, the new version cannot do significantly worse than the old version and in many

Table 7-1: Two exact versions of the algorithm

Problem	W/o Improvement	With Improvement
SCH10	19.1	9.6
SCH14	>300	42.4
ALB11	4.8	5.1
ALB20	18.2	19.6

cases will do much better.

All computation times cited in this chapter are from runs on a DEC-20 at Stanford University. They include internal set up time but not time spent reading in data. All times are in CPU seconds. The code used is highly experimental.

- **Worst Case Bounds**

In the case of a general binary problem with n variables, an enumerative algorithm must consider, implicitly or explicitly, 2^n possible solutions. In this general problem the variables could be partitioned into m sets, containing n_i variables for $i = 1, 2, \ldots, m$. Then $n = \sum_{i=1}^{m} n_i$ and $2^n = \prod_{i=1}^{m} 2^{n_i}$. In the MCIP, where such a partitioning is natural, the objective is not simply to find any combination of zeros and ones that are feasible, but to choose one variable from each GUB-set to assign the value one, so that the configuration is feasible in the general constraints. The number of possible combinations is then $\prod_{i=1}^{m} n_i$. Clearly, for any positive integers m and $n_i, i = 1, 2, \ldots, m$, it is true that

$$\prod_{i=1}^{m} n_i < \prod_{i=1}^{m} 2^{n_i}.$$

In general binary trees, the number of branches and the maximum number of solutions investigated, is a function only of the number of variables. In the multiple choice case the number of branches is a function of both the number of variables and the configuration of the GUB partitioning. Following is a discussion of three configurations: the worst for computation, an average case, and the best for computation.

Given n variables, the worst possible configuration is $n/2$ GUB–sets each with 2 elements. In this case the upper bound on investigated solutions is

$$\prod_{i=1}^{n/2} 2 = 2^{n/2} \approx 1.4^n.$$

Though this is clearly better than the 2^n upper bound on the general problem, it is exponential and greatly limits the size of problems that are solvable. Figure 7–1 shows a plot of the number of branches versus the number of variables for this configuration.

In Chapter 3 it was mentioned that any binary problem could be formulated as a MCIP by introducing the complements for each variable and forming GUB–sets from each variable and its complement. If n is the number of variables in the original binary program, then this MCIP formulation has $2n$

variables. The upper bound on branches investigated for this configuration is then $2^{(2n)/2} = 2^n$. As mentioned in Chapter 3, this is identical to the upper bound for general binary formulations.

An average case configuration is "square," that is,

$$m = n_1 = n_2 = \cdots = n_m.$$

The classical assignment problem and the scheduling problem shown in Chapter 4 are examples of "square" problems.

The upper bound on investigated solutions for this configuration is

$$\prod_{i=1}^{m} m = m^m.$$

Since the number of variables is $n = m^2$, the upper bound on number of partial solutions explicitly investigated is $\sqrt{n}^{\sqrt{n}}$.

Theorem 7-1: The worst case bound on number of possible solutions explicitly investigated for the "square" case is better than exponential.

Proof: For the bound to be exponential, there would exist some $p > 1$ such that

$$\sqrt{n}^{\sqrt{n}} \geq p^n, \quad \forall n \geq N,$$

where N is some large constant. If this were true, then for $n \geq N$

$$\sqrt{n}\ln(\sqrt{n}) \geq n \ln p$$

since the logarithm function is increasing. This implies that

$$\ln p \leq \frac{\ln(\sqrt{n})}{\sqrt{n}},$$

assuming that $n > 0$. The exponential is also an increasing function so, taking the exponential of both sides gives

$$p \leq \exp(\frac{\ln(\sqrt{n})}{\sqrt{n}}) = \exp(\frac{\ln n}{2\sqrt{n}}).$$

Since it is true that

$$\lim_{n\to\infty} \frac{\ln n}{2\sqrt{n}} = 0,$$

$$\lim_{n\to\infty} \exp(\frac{\ln n}{2\sqrt{n}}) = 1.$$

Consequently, for any $p > 1$, there exists n^* such that for $n \geq n^*$,

$$\sqrt{n}^{\sqrt{n}} < p^n.$$

Hence, the bound is better than exponential.∎

Theorem 7–2: This bound is not polynomial.

Proof: **If the bound were polynomial, then some $a \in \Re_+$ and $l \in Z$ could be found such that**

$$\sqrt{n}^{\sqrt{n}} \leq an^l, \forall n \in Z.$$

Since the logarithm function is increasing it would then be true that

$$\sqrt{n}\ln(\sqrt{n}) \leq \ln a + 2l\ln(\sqrt{n}).$$

Since $n \geq 1$, this would imply that

$$\sqrt{n} \leq \frac{\ln a}{\ln\sqrt{n}} + 2l, \forall n \in Z.$$

This is clearly false. Hence, the bound is not polynomial.∎

The last configuration to be analyzed is that having a fixed number of GUB–sets. Let m be held constant as the number of variables is increased, that is, all new variables are added to existing GUB–sets rather than introducing new GUB–sets. The number of elements in any particular GUB–set is not greater than the total number of variables so

$$\prod_{i=1}^{m} n_i \leq \prod_{i=1}^{m} n = n^m.$$

Hence, the upper bound for this configuration is polynomial with order, at most, equal to the number of GUB–sets.

The upper bound on the number of solutions explicitly investigated by this algorithm is very dependent on the GUB configuration of the variables. In some cases it is exponential, in some cases polynomial and in some cases neither. In practice the algorithm has shown only polynomial behavior in limited testing. Details are given later in this chapter.

- **Sorting**

Sorting is very important to the computational efficiency of this algorithm. At the beginning of the algorithm the costs corresponding to each GUB-set must be sorted. Also, after each step the costs associated with the current position must be sorted in order to map each GUB-set to an order. Choosing a sorting technique deals with a well studied class of problems but becomes interesting because the best technique for use in this algorithm, the bubble sort, is considered to be one of the poorer methods in general ([Knuth, 1973-1]).

Most of the sorting during the algorithm comes when the costs associated with a current position are sorted during each iteration. This sorting is needed because each step changes the current position which changes the associated costs. The first time this is done, the list to be sorted begins in "random" order. Thereafter only one element of the list is out of place, since the step only changes one component of the position vector, that mapped to the order of the step. The other components are unchanged, so, the relationships of these elements remain unchanged. The sorting task reduces to finding the correct position of the cost of this new component in the list. Because of the initial ordering of the costs associated with each GUB-set and the fact that a step moves the position further along in some GUB-set, it is also known that the new component will move down in the list from the position of its

Table 7–2: Random Element Sorting

List Size	Heapsort	Bubble Sort
50	.14	.10
100	.20	.13
500	.93	1.19
1000	1.88	4.50
2000	3.98	19.18

predecessor, if it moves at all.

The bubble sort has complexity n^2, that is, given a randomly ordered list of n elements it will take on the order of n^2 comparisons to sort the list. This is inferior to sorting techniques such as the Shell sort or the heapsort, which have complexity $n \ln n$ [Knuth, 1973–1]. The bubble sort goes through the list comparing consecutive elements. When it finds two elements out of order, relative to each other, it switches their positions. For example, suppose one wished to sort the list (5,4,1,3) to "lowest first." The bubble sort would compare 5 and 4 and, finding them out of order, switch their positions to obtain $(4, 5, 1, 3)$. The next pair, 5 and 1, then is tested. They are out of order so the list becomes (4,1,5,3). Another comparison and switch leaves (4,1,3,5). Now the entire process is repeated until all elements are in the correct position.

Table 7–3: Resorting One Element

List Size	Heapsort	Bubble Sort
50	.14	.10
100	.24	.14
500	.96	.26
1000	2.08	.57
2000	4.12	1.03

The efficiency of this technique is improved by the realization that after the first full pass, the largest element is certain to have passed to the end of the list. It need not be considered again. After the second pass, the second largest element will be in its correct position. Each pass, the list left to consider becomes one element shorter. This, however, does not reduce the complexity of the process.

Table 7–2 shows a comparison of run times for the bubble sort routine used in the algorithm and a heapsort routine from [Knuth, 1973–1] on randomly generated lists of elements from a uniform [0,100] distribution. The lists in this comparison began in random order. As expected, when the number of elements in the list grows, the heapsort performs much better. Table 7–3 shows a comparison of run times for the same two routines. The difference here is that the lists began in correct order with the exception of one randomly chosen element in the list which is increased by a value chosen randomly from

Table 7–4: All League Scheduling Problems

Teams	Variables	Partial Solutions	Run Time
10	100	116.4	12.1
12	144	59.6	9.9
14	196	154.6	24.4
16	256	140.8	31.6
18	324	285.4	73.6

a uniform [0,25] distribution. This test was constructed to simulate the lists that are sorted during the process of the algorithm. For these problems the heapsort performs almost exactly the same as it did on the general lists. The bubble sort, however, does much better in this case. This occurs because the bubble sort now makes only one pass. On this type of list the bubble sort has complexity n. Because of these tests, the bubble sort is used in this code of the algorithm.

- **Scheduling Problems**

The algorithm was tested on a sequence of 25 athletic league scheduling problems like that described in Chapter 4. The cities used for these problems were: Atlanta, Boston, Chicago, Cleveland, Denver, Detroit, Houston, Indianapolis, Kansas City, Los Angeles, Milwaukee, Newark, New York, Philadelphia, Phoenix, Portland, Salt Lake City, San Antonio, San Diego,

Table 7–5: First Inspection League Scheduling Problems

Teams	Variables	Partial Solutions	Run Time
12	144	0	3.6
14	196	0	5.7
16	256	0	9.1
18	324	0	13.9

San Francisco, Seattle, and Washington, D. C. For each size category five problems were run. Each run was made by *randomly choosing* the appropriate number of teams from the above list and considering them the "league." The averages, in each size category, of run times and number of partial solutions investigated is shown in table 7–4.

Determining the rate at which run times increase as a function of the number of variables, is difficult due to the high variance in sample times. In many problems of this type the algorithm solves the problem on the first inspection. In this case the only computation involved is solving one surrogate constraint LP. As seen in table 7–5, when this occurs, the run times increase approximately linearly in the number of variables. This solution by inspection happened in 9 of the 25 runs.

When the first inspection fails to solve the problem, the number of partial solutions investigated increases approximately linearly in the number of variables. However, as the problems get larger, the computation time

Table 7–6: Long League Scheduling Problems

Teams	Variables	Partial Solutions	Run Time
10	100	116.4	12.1
12	144	149.0	19.4
14	196	257.7	36.9
16	256	352.0	65.4
18	324	356.8	88.5

for each iteration also increases approximately linearly in the number of variables. Hence, the run times for scheduling problems that are not solved in the first inspection increases approximately quadratically in the number of variables. Table 7–6 shows results of runs of this type. Figure 7–2 shows plots of average times for first inspection solutions, long solutions and the combined sample. (For this and all subsequent figures, the curves plotted through the scatter points are least squares fits.) Figure 7–3 shows the same set of plots for the number of partial solutions investigated.

- **Assembly Line Balancing Problems**

An analysis also was done of the performance of the algorithm on four assembly line balancing problems, similar to that described in Chapter 4. The problems consisted of 4, 7, 11 and 20 tasks. The 4 task problem is that solved by hand in Chapter 4. The other problems are parts of the example

Table 7-7: Assembly Line Balancing Problems

Tasks	Variables	Partial Solutions	Run Time
4	16	8	.4
7	42	9	1.8
11	66	16	5.1
20	120	25	19.6

in [Kilbridge & Wester, 1962-1]. For example, the 20 task problem consists of elements 26 through 45 of the Kilbridge & Wester example.

The data for these runs are presented in table 7-7, and plots of time and iterations can be found in figures 7-4 and 7-5.

The number of iterations required to solve a problem of this type increases approximately linearly with the number of tasks. The amount of time required to complete an iteration also increases approximately linearly. The total run time increases approximately quadratically in the number of tasks.

Reviews of assembly line balancing can be found in [Mastor, 1970-1], and [Buffa, 1977-1]. In practice, assembly line balancing problems generally are solved heuristically. Real problems tend to be too large for existing exact codes. The results cited in table 7-7 are from a code that uses no special characteristic of the assembly line balancing problem other than its GUB structure. If designed specially for this problem, the results in table 7-7 suggest that it may be an efficient technique for solving these problems. By using the standard heuristic of grouping tasks, this technique could become competitive with existing heuristic methods for solving large balancing problems. This is an area targeted for future research.

It would be desirable to compare the performance of this algorithm and those in [Mevert & Suhl, 1977-1] and [Sinha & Zoltners, 1979-1]. However, the former paper was done on a proprietary basis so that details of implementation and most of the test problems reported are not available. The one problem cited from the literature was from [Kilbridge & Wester, 1962-1]. This problem was used to generate the larger assembly line balancing problems cited in table 7-7. However, due to the small capacity of the computer used for this dissertation, the complete problem was not run. The latter paper presents virtually no computational results and only a very sketchy description of the algorithm. When more information is available on this algorithm a comparison may be carried out.

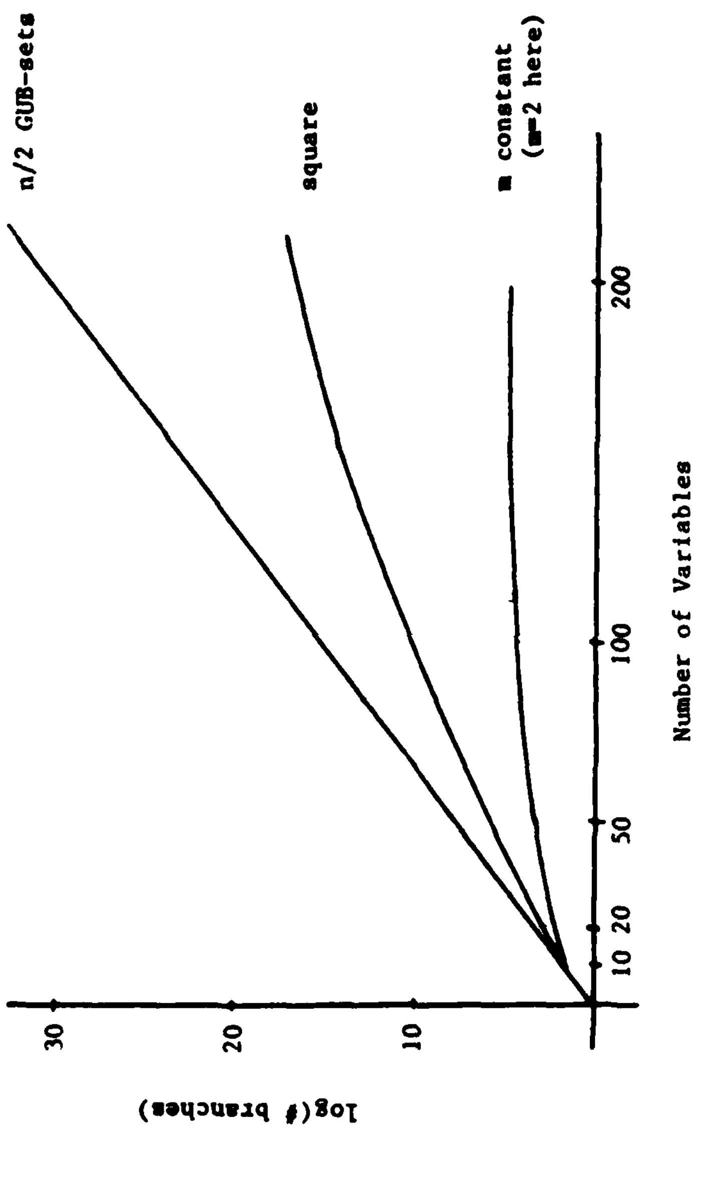


FIGURE 7-1: Worst Case Bounds for Three Configurations.

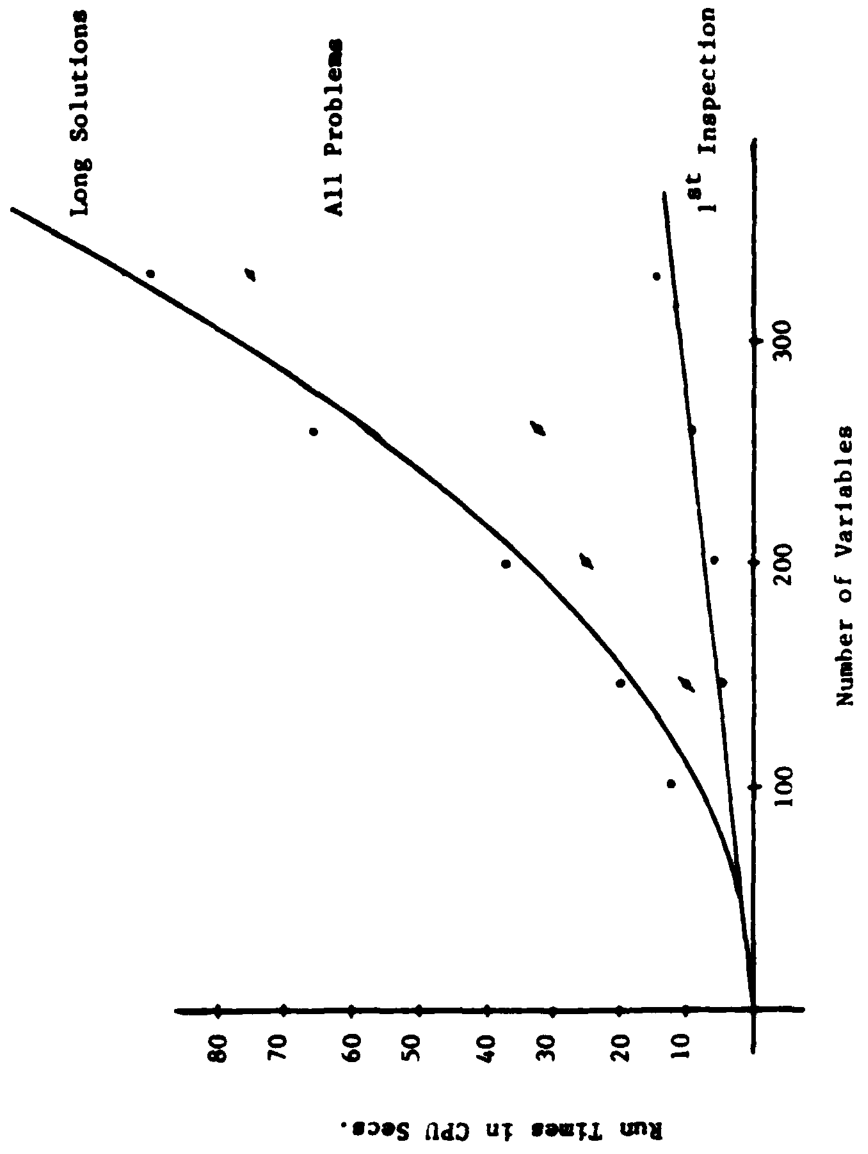


FIGURE 7-2: Run Times vs. # Variables for the Scheduling Problems.

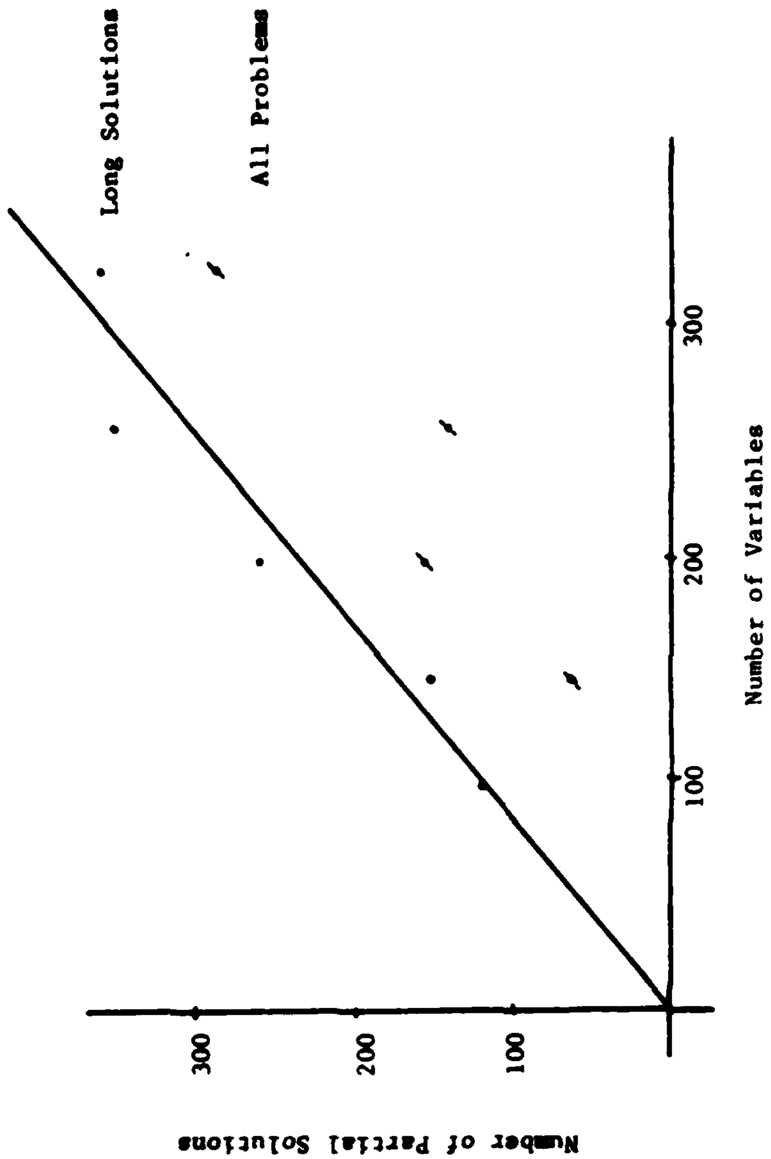


FIGURE 7-3: # Partial Solutions vs. # Variables for the Scheduling Problems.

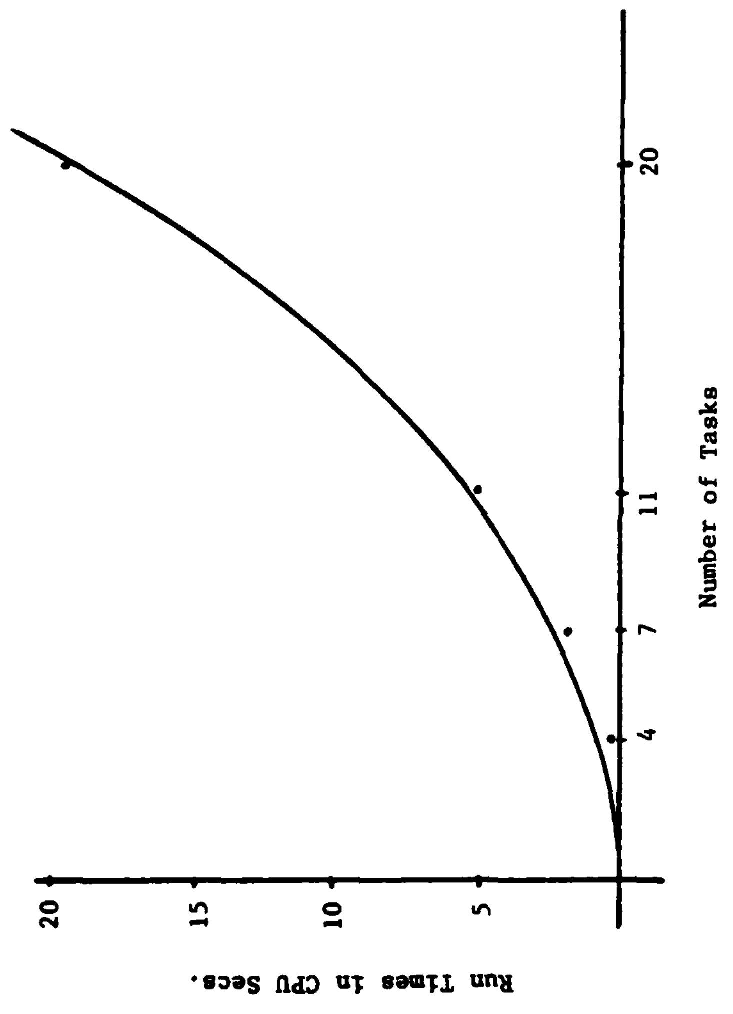


FIGURE 7-4: Run Times vs. Number of Tasks for the Assembly Line Balancing Problems.

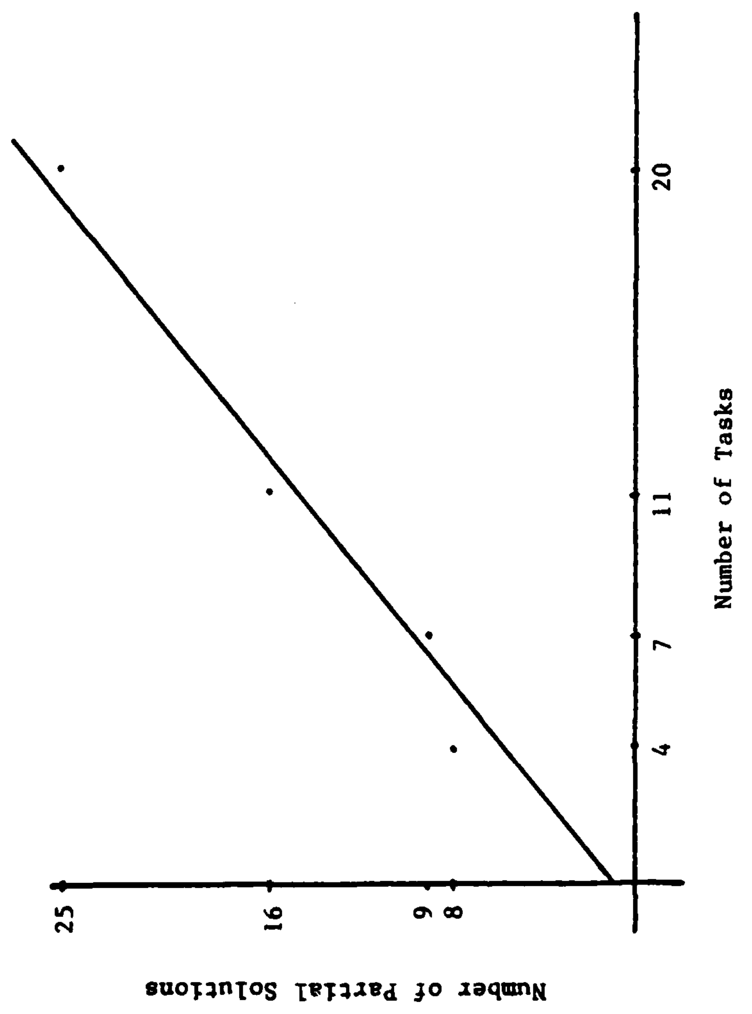


FIGURE 7-5: # Partial Solutions vs. # Tasks for the Assembly Line Balancing Problems.

CHAPTER 8

CONCLUSIONS

The MCIP constitutes an important class of problems having many real applications. Though work on similar problems dates back to the early sixties, work on this specific problem has begun only in the last few years.

The most important question to be answered is whether or not the algorithm presented in this dissertation is an efficient way to solve Multiple Choice Integer Programs. It is rarely possible to answer questions of this type definitively and certainly it cannot be done in this case. From the evidence uncovered in this dissertation it seems that this algorithm shows promise for at least the two problems tested, league scheduling and assembly line balancing. There is no reason to believe that this result will not generalize to many other applications. To determine how the algorithm performs relative to other methods for solving these problems it will be necessary to develop a more streamlined code on a faster machine and do much more extensive testing. What has been demonstrated here is that run times increase approximately quadratically in the size of the problem for the two classes of problems tested.

The algorithm can be run in exact or heuristic mode. The exact version has been proven to find an optimal solution in finite time. The heuristic version has guaranteed accuracy (upper bound on % error), and in all testing found a true optimal solution.

Rather than using only the exact mode to find an optimal solution, the algorithm tends to run faster by making two passes: one heuristic pass followed by an exact pass over the last positions considered in the heuristic pass. This still achieves optimality because the optimal solutions can be proven to have been skipped over at the end of the heuristic pass if at all.

It is very important in this algorithm to employ a surrogate constraint LP package. Three possible LP formulations are presented for surrogate constraints on the MCIP. The formulation used in the code for this algorithm was chosen because it requires only primal pivots. This is important here since the LP package used in this code (LPM-1) stores the constraint coefficients compactly, making dual pivots inefficient.

An alternative formulation, GLP_S–3, would be superior if an LP package were used that avoided the compact storage difficulty. Bases are much smaller in this formulation, leading to time savings in solving surrogate constraint LP's.

Possible future research in this area includes implementing the alternative surrogate constraint LP formulation and streamlining the code. Many application areas mentioned in Chapter 4 can then be investigated closely to determine which are solved most efficiently by this algorithm.

Like some versions of the additive algorithm for general binary programming, this algorithm depends on cost ordering to choose separations and descendants. Separations are made by fixing variables that add the least cost. However, many other criteria have been tried in the general case. One example is separation by fixing the variable that decreases the current infeasibility most. These approaches should also be tried on the MCIP.

The importance of the MCIP lies in the wide range of real problems that can be formulated as MCIP's. The true test of this or any other algorithm proposed for this class of problems will be its performance on real problems.

BIBLIOGRAPHY

Armstrong, R. D. and J. L. Balinfty (1975), "A Chance Constrained Multiple Choice Algorithm," *Operations Research*, Vol. 23, pp. 494–510.

Ashour, S. and A. R. Char (1970), "Computational Experience on Zero–One Programming Approach to Various Combinatorial Problems," *Journal of the Operations Research Society of Japan*, Vol. 13, pp. 78–108.

Baker, K. (1974), *Introduction to Sequencing and Scheduling*, Wiley, New York.

Balas, Egon (1965), "An Additive Algorithm for Solving Linear Programs with Zero–One Variables," *Operations Research*, Vol. 13, pp. 517–546.

Ball, B. C. and D. B. Webster (1977), "Optimal Scheduling for Even–Numbered Team Athletic Conferences," *AIIE Transactions*, Vol. 9, pp. 161–169.

Barr, R. S., F. Glover and D. Klingman (1977), "A New Alternating Basis Algorithm for Semi–Assignment Networks," *Management Science, Report Series* No. 77–3, University of Colorado.

Beale, E. M. L. and J. A. Tomlin (1970), "Special Facilities in a General Mathematical Programming System for Non–convex Problems Using Ordered Sets of Variables," *Proceedings of the Fifth IFORS Conference*.

Bean, J. C. and J. R. Birge (1980), "Reducing Travelling Costs and Player Fatigue in the NBA," *Interfaces*, Vol. 10, pp. 98–102.

Bellmore, M. and G. L. Nemhauser (1968), "The Traveling Salesman Problem: A Survey," *Operations Research*, Vol. 16, pp. 538–558.

Bricker, D. L. (1977), "Reformulation of Special Ordered Sets for Implicit Enumeration Algorithms, with Applications in Nonconvex Separable Programming," *AIIE Transactions*, Vol. 9, pp. 195–203.

Brown, A. R. (1969), "Selling Television Time: An Optimization Problem," *Computer Journal*, Vol. 12, pp. 201–207.

BIBLIOGRAPHY

Buffa, E. (1977), *Modern Production Management* Wiley, New York.

Conway, R. W., W. L. Maxwell and L. W. Miller (1967), *Theory of Scheduling,* Addison-Wesley, Palo Alto.

Dantzig, G. B. (1963), *Linear Programming and Extentions.* Princeton University Press. Princeton, N. J.

Dantzig, G. B. (1960), "On the Significance of Solving Linear Programming Problems with Some Integer Variables," *Econometrica,* Vol. 28, pp. 30–44.

Dantzig, G. B. and R. M. Van Slyke (1964), "A Generalized Upper-Bounded Technique for Linear Programming," *Proceedings of the IBM Scientific Computing Symposium on Combinatorial Problems.*

Dantzig, G. B. and R. M. Van Slyke (1967), "Generalized Upper Bounding Techniques," *Journal of Computer and System Science,* Vol. 1, pp. 213–226.

Fleischman, Bernhard (1967), "Computational Experience with the Algorithm of Balas," *Operations Research,* Vol. 15, pp. 153–155.

Freeman, R. J. (1965), "Computational Experience With the Balas Integer Programming Algorithm," Rand paper P-3241.

Geoffrion, A. M. (1969), "An Improved Implicit Enumeration Approach for Integer Programming," *Operations Research,* Vol. 17, pp. 437–453.

Geoffrion, A. M. and R. E. Marsten (1972), "Integer Programming Algorithms: A Framwork and State-of-the-Art Survey," *Management Science,* Vol. 18, pp. 465- 491.

Geoffrion, A. M. (1967), "Integer Programming by Implicit Enumeration and Balas' Method," *SIAM Review* , Vol. 9, pp. 178–190.

Greenberg, H. and R. L. Hegerich (1970), "A Branch Search Algorithm for the Knapsack Problem," *Management Science,* Vol. 16, pp. 327–332.

Healy, W. C. (1964), "Multiple Choice Programming," *Operations Research*, Vol. 12, pp. 122–138.

Hillier, F. S. and G. J. Lieberman (1980), *Operations Research*, 3rd ed., Holden Day, San Francisco.

Ibaraki, T. (1976), "Computational Efficiency of Approximate Branch and Bound Algorithms," *Mathematics of Operations Research*, Vol. 1, pp. 287–298.

Ignall, E. J. (1965), "A Review of Assembly Line Balancing," *The Journal of Industrial Engineering*, Vol. 16, pp. 194–210.

Johnson, M. A., A. Zoltners and P. Sinha (1979), "An Allocation Model for Catalogue Space Planning," *Management Science*, Vol. 25, pp. 117–129.

Kilbridge, M. D. and L. Wester (1962), "A Review of Analytical Systems of Line Balancing," *Operations Research*, Vol. 10, pp. 626–638.

Knuth, D. (1973), *The Art of Computer Programming Volume 3: Sorting and Searching*, Addison–Wesley, Menlo Park.

Kuhn, H. W. (1955), "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly*, Vol. 2, pp. 83–97.

Land, A. H. and A. G. Doig (1960), "An Automatic Method of Solving Discrete Programming Problems," *Econometrica*, Vol. 28, pp. 497–520.

Lawler, E. L. and D. E. Wood (1966), "Branch and Bound Methods: A Survey," *Operations Research*, Vol. 16, pp. 699–719.

Lawrie, N. L. (1969), "An Integer Programming Model of a School Time Tabling Problem," *Computer Journal*, Vol. 12, pp. 307–316.

Mastor, A. A. (1970), "An Experimental Investigation and Comparative Evaluation of Production Line Balancing Techniques," *Management Science*, Vol. 16, pp. 728–746.

Mevert, P. and U. Suhl (1977), "Implicit Enumeration with Generalized Upper

Bounds," *Annals of Discrete Mathematics 1*, pp. 393-402.

Nauss, Robert M. (1976), "An Efficient Algorithm for the 0–1 Knapsack Problem," *Management Science*, Vol. 23, pp. 27–31.

Nauss, R. M. (1975), "The 0–1 Knapsack Problem with Multiple Choice Constraints," Working Paper, U. Missouri–St. Louis, March 1975, Revised May 1976

Peterson, Clifford C. (1967), "Computational Experience with Variants of the Balas Algorithm Applied to the Selection of R&D Projects," *Management Science*, Vol. 13, pp. 736–750.

Pfefferkorn, C. E. and J. A. Tomlin (1976), "Design of a Linear Programming System for the Illiac IV," Technical Report SOL 76–8, Systems Optimization Lab, Stanford University.

Reinfeld, N. V. and W. R. Vogel (1958), *Mathematical Programming*, Prentice–Hall, Englewood Cliffs, N. J.

Ross, G. Terry and Richard M. Soland (1973), "A Branch and Bound Algorithm for the Generalized Assignment Problem," *Mathematical Programming* No. 8, pp. 91–103.

Ross, G. Terry and Richard M. Soland (1977), "Modeling Facility Locations Problems as Generalized Assignment Problems," *Management Science*, Vol. 24, pp. 345–357.

Ross, G. Terry and Andris Zoltners (1979), "Weighted Assignment Models and Their Applications," *Management Science*, Vol. 25, pp. 683–696.

Sinha, P. and A. Zoltners (1979), "A Multiple–Choice Integer Programming Algorithm," Graduate School of Management, Northwestern University.

Sinha, P. and Andris Zoltners (1979), "The Multiple Choice Knapsack Problem," *Operations Research*, Vol. 27, pp. 503–515.

Thangavelu, S. R. and C. M. Shetty (1971), "Assembly Line Balancing by Zero–One Integer Programming," *AIIE Transactions*, Vol. 3, pp. 61–68.

Trauth, C. A., Jr. and R. E. Woolsey (1969), "Integer Linear Programming: A Study in Computational Efficiency," *Management Science*, Vol. 15, pp. 481–493.

Westley, G. D. (1980), "Project Analysis in the Context of National Planning, With Case Application in Latin America," *Papers on Project Analysis* No. 14, Inter–American Development Bank.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 96	2. GOVT ACCESSION NO. AD-A092 691	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) AN ADDITIVE ALGORITHM FOR THE MULTIPLE CHOICE INTEGER PROGRAM		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL REPORT
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) JAMES CARL BEAN		8. CONTRACT OR GRANT NUMBER(s) Nooo14-76-C-0418
9. PERFORMING ORGANIZATION NAME AND ADDRESS OPERATIONS RESEARCH PROGRAM -ONR DEPARTMENT OF OPERATIONS RESEARCH STANFORD UNIVERSITY, STANFORD, CALIF.		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (NR-047-061)
11. CONTROLLING OFFICE NAME AND ADDRESS OFFICE OF NAVAL RESEARCH OPERATIONS RESEARCH PROGRAM CODE 434 ALRINGTON, VA. 22217		12. REPORT DATE OCTOBER 1980
		13. NUMBER OF PAGES 123
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited. Reproduction in whole or in part is permitted for any purpose of the United States Government.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Also issued as Technical Report No. 80-26, Dept. of Operations Research - Stanford University under NSF Grant MCS76-81259.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

INTEGER PROGRAMMING, MULTIPLE CHOICE INTEGER PROGRAMMING, ALGORITHMS, ADDITIVE ALGORITHM

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

PLEASE SEE OTHER SIDE.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TECHNICAL REPORT NO. 96 Author: James C. Bean

The problem dealt with in this report is the Multiple Choice Integer Program (MCIP), a special case of the integer linear programming problem with binary variables. In this problem the variables are partitioned (usin the classical definition of "partition"). Exactly one variable in each partitioning set has the value one in any feasible solution and all other variables have the value zero. The problem also allows any additional constraints that can be expressed as linear inequalities. The objective is to choose elements from these sets so as to minimize cost.

Chapter 2 discusses the history of this problem and solution technique developed for it and similar problems such as the classical assignment problem, the generalized assignment problem, the multiple choice problem, the generalized upper bounding problem and the multiple choice knapsack problem. Chapter 3 presents discussion of additive type algorithms and the algorithm proposed in this report. It also includes definitions and notations.

Applications of this algorithm follow in Chapter 4 with worked example for three problems: a scheduling problem, an assembly line balancing problem and a distribution problem. In Chapter 5 it is proven that the algorithm will correctly solve any problem of this type in finite time.

In Chapter 6 Geoffrion's surrogate constraints are revised to take advantage of the additonal structure in this problem. Three possible formulations of surrogate constraint LP's are presented. By taking advantage of the special structure of the MCIP, the size of the LP's relative to those used for general binary programming is greatly reduced.

A heuristic version of the algorithm is developed in Chapter 7 that has guaranteed accuracy (upper bound on % error) of the user's choice. In all tests, this heuristic never failed to find a true optimal solution. This heuristic is evolved into a faster exact version of the algorithm. This chapter also includes an analysis of worst case bounds on performance, sorting techniques and the results of experimental runs on two classes of problems: league scheduling problems and assembly line balancing problems. For both of these classes of problems, the computer times required to solve problems increases approximately quadratically in the size of the problem.

96/80-26